DOCUMENT RESUME

ED 143 506 SE 022 983

AUTHOR Charp, Sylvia; And Others
TITLE Algorithms, Computation and Mathematics. Student Text. Revised Edition.
INSTITUTION Stanford Univ., Calif. School Mathematics Study Group.
SPONS AGENCY National Science Foundation, Washington, D.C.
PUB DATE 66
NOTE 456p.; For related documents, see SE 022 984-988; Not available in hard copy due to marginal legibility of original document; Pages 3-6 missing; Best Copy Available

EDRS PRICE MF-$0.83 Plus Postage. HC Not Available from EDRS.
DESCRIPTORS *Algorithms; *Computers; *Instructional Materials; Programing Languages; Secondary Education; *Secondary School Mathematics; *Textbooks
IDENTIFIERS *School Mathematics Study Group

ABSTRACT

This text contains material designed for about 18 weeks of study at grades 11 or 12. Use of a computer with the course is highly recommended. Developing an understanding of the relationship between mathematics, computers, and problem solving is the main objective of this book. The following chapters are included in the book: (1) Algorithms, Language, and Machines; (2) Input, Output, and Assignment; (3) Branching and Subscripted Variables; (4) Looping; (5) Functions and Procedures; (6) Approximations; (7) Some Mathematical Applications; and (8) Compilation and Some Other Non-Numeric Problems. Also included is a discussion on future computer applications. (RH)

****
* Documents acquired by ERIC include many informal unpublished *
* materials not available from other sources. ERIC makes every effort *
* to obtain the best copy available. Nevertheless, items of marginal *
* reproducibility are often encountered and this affects the quality *
* of the microfiche and hardcopy reproductions ERIC makes available *
* via the ERIC Document Reproduction Service (EDRS). EDRS is not *
* responsible for the quality of the original document. Reproductions *
* supplied by EDRS are the best that can be made from the original. *
****

ED143506

U S DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
NATIONAL INSTITUTE OF
EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

SMSG

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC) AND THE ERIC SYSTEM CONTRACTORS."

ALGORITHMS, COMPUTATION AND MATHEMATICS

Student Text

Revised Edition

The following is a list of all those who participated in the preparation of this volume:

Sylvia Charp, Dobbins Technical High School, Philadelphia, Pennsylvania
Alexandra Forsythe, Gunn High School, Palo Alto, California
Bernard A. Galler, University of Michigan, Ann Arbor, Michigan
John G. Herriot, Stanford University, California
Walter Hoffmann, Wayne State University, Detroit, Michigan
Thomas E. Hull, University of Toronto, Toronto, Ontario, Canada
Thomas A. Keenan, University of Rochester, Rochester, New York
Robert E. Monroe, Wayne State University, Detroit, Michigan
Silvio O. Navarro, University of Kentucky, Lexington, Kentucky
Elliott I. Organick, University of Houston, Houston, Texas
Jesse Peckenham, Oakland Unified School District, Oakland, California
George A. Robinson, Argonne National Laboratory, Argonne, Illinois
Phillip M. Sherman, Bell Telephone Laboratories, Murray Hill, New Jersey
Robert E. Smith, Control Data Corporation, St. Paul, Minnesota
Warren Stenberg, University of Minnesota, Minneapolis, Minnesota
Harley Tillitt, U. S. Naval Ordnance Test Station, China Lake, California
Lyneve Waldrop, Newton South High School, Newton, Massachusetts

The following were the principal consultants:

George E. Forsythe, Stanford University, California
Bernard A. Galler, University of Michigan, Ann Arbor, Michigan
Wallace Givens, Argonne National Laboratory, Argonne, Illinois

SE 022 983

© 1965 and 1966 by The Board of Trustees of the Leland Stanford Junior University
All rights reserved
Printed in the United States of America

Permission to make verbatim use of material in this book must be secured from the Director of SMSG. Such permission will be granted except in unusual circumstances. Publications incorporating SMSG materials must include both an acknowledgment of the SMSG copyright (Yale University or Stanford University, as the case may be) and a disclaimer of SMSG endorsement. Exclusive license will not be granted save in exceptional circumstances, and then only by specific action of the Advisory Board of SMSG.

Financial support for the School Mathematics Study Group has been provided by the National Science Foundation

FOREWORD

The increasing contribution of mathematics to the culture of the modern world, as well as its importance as a vital part of scientific and humanistic education, has made it essential that the mathematics in our schools be both well selected and well taught.

With this in mind, the various mathematical organizations in the United States cooperated in the formation of the School Mathematics Study Group (SMSG). SMSG includes college and university mathematicians, teachers of mathematics at all levels, experts in education, and representatives of science and technology. The general objective of SMSG is the improvement of the teaching of mathematics in the schools of this country. The National Science Foundation has provided substantial funds for the support of this endeavor.

One of the prerequisites for the improvement of the teaching of mathematics in our schools is an improved curriculum -- one which takes account of the increasing use of mathematics in science and technology and in other areas of knowledge and at the same time one which reflects recent advances in mathematics itself. One of the first projects undertaken by SMSG was to enlist a group of outstanding mathematicians and mathematics teachers to prepare a series of textbooks which would illustrate such an improved curriculum.

The professional mathematicians in SMSG believe that the mathematics presented in this text is valuable for all well-educated citizens in our society to know and that it is important for the precollege student to learn in preparation for advanced work in the field. At the same time, teachers in SMSG believe that it is presented in such a form that it can be readily grasped by students.

In most instances the material will have a familiar note, but the presentation and the point of view will be different. Some material will be entirely new to the traditional curriculum. This is as it should be, for mathematics is a living and an ever-growing subject, and not a dead and frozen product of antiquity. This healthy fusion of the old and the new should lead students to a better understanding of the basic concepts and structure of mathematics and provide a firmer foundation for understanding and use of mathematics in a scientific society.

It is not intended that this book be regarded as the only definitive way of presenting good mathematics to students at this level. Instead, it should be thought of as a sample of the kind of improved curriculum that we need and as a source of suggestions for the authors of commercial textbooks. It is sincerely hoped that these texts will lead the way toward inspiring a more meaningful way of teaching Mathematics, the Queen and Servant of the Sciences.

PREFACE

To The Student:

A new computer science is emerging as a discipline in the colleges and universities of our land. On a growing number of college campuses students who are majoring in engineering, mathematics, business, and other areas, are being urged or even required to gain at least an introductory view of this new science before stepping into a computer-influenced society.

An introduction to computer science is much more than a quick "how to do it" on the use of computers. Among other things, it is based on and is an extension of the mathematics you now know. Developing an understanding of the relationship between mathematics, computers, and problem solving is the main objective of this book. The "how-to-do-it" or technique of computing is also necessary. You will find yourself acquiring such skills as a valuable by-product of the learning process that you are now embarking on. The opportunity is here. Seize it!

TABLE OF CONTENTS

FOREWORD

PREFACE

Chapter

1 ALGORITHMS, LANGUAGE AND MACHINES 1
1-1. Introduction. 1
1-2. A Little History. 3
1-3. Some Technical Aspects of Computers 7
1-4. Numbers and Other Characters. 17
1-5. Algorithms. 24
1-6. Comments on Language. 31

2 INPUT, OUTPUT AND ASSIGNMENT. 35
2-1. The Flow Chart Concept. 35
2-2. Repetition. 41
2-3. Assignment and Variables. 45
2-4. Arithmetic Expressions. 54
2-5. Rounding Functions. 69
2-6. Alphanumeric Data 81

3 BRANCHING AND SUBSCRIPTED VARIABLES 89
3-1. Branching 89
3-2. Auxiliary Variables 102
3-3. Compound Conditions and Multiple Branching. 120
3-4. Precedence Levels for Relations 130
3-5. Subscripted Variables 134
3-6. Double Subscripts 149

4 LOOPING 157
4-1. Looping 157
4-2. Illustrative Examples 165
4-3. Table-Look-Up 179
4-4. Nested Loops. 190

5 FUNCTIONS AND PROCEDURES
5-1. Reference Flow Charts 215
5-2. Mathematical Functions. 221
5-3. Getting In and Out of a Functional Reference. 229
5-4. Procedures. 241
5-5. Extensions to Reference Flow Charts and their Models. . . . 251
5-6. Character Strings 259

6 APPROXIMATIONS. 265
6-1. Introduction. 265
6-2. Chopping and Rounding to n Digits 267
6-3. Three Digit Arithmetic. 268
6-4. Implications of Finite Word Length. 271
6-5. Non-Associativity of Computer Arithmetic. 277
6-6. Some Pitfalls 281
6-7. More Pitfalls 286
6-8. Approximating Functions 288

Chapter

7 SOME MATHEMATICAL APPLICATIONS. 295
[illegible]-1. Root of an Equation by Bisection. 295
7-2. The Area Under a Curve: An example, $y = 1/x$ between $x = 1$ and $x = 2$. . . 311
7-3. The Area Under a Curve: The General Case 324
7-4. Simultaneous Linear Equations: Developing a systematic method of solution. 330
7-5. Simultaneous Linear Equations: Gauss Algorithm . . . 338

8 COMPILATION AND SOME OTHER NON-NUMERIC PROBLEMS 359
8-1. Introduction. 359
8-2. Symbol Manipulation 361
8-3. A Language to be Translated 369
8-4. Prescan (the preliminary steps of a compiler) 374
8-5. The Decomposition of Assignment Statements. 390
8-6. A Decomposition Flow Chart. 401

Epilogue THE FUTURE FOR COMPUTERS. 409
E-1. Computer Applications Today 409
E-2. Changes in Computer Directions. 411
E-3. New Problems. 412
E-4. Preparing for the Future. 414

APPENDIX A. 415

APPENDIX B. 446

INDEX . .

Chapter I

ALGORITHMS, LANGUAGE AND MACHINES

1-1 Introduction

These are the early, exciting years of a revolution in man's ability to process and use information. This is the computer revolution. It promises to be at least as far-reaching as the industrial revolution of the steam engine. Twenty-five thousand electronic computers are in use in the United States alone. They represent an investment of eight billion dollars. What's more, six to seven thousand computers are being made each year.

The work of literally millions of people (including scientists, engineers, economists, medical doctors, nurses, designers, salesmen, teachers, machinists, financial workers, social workers, writers, editors, linguists, archeologists, etc.) is being changed by new uses of computers. Every person in the country needs some understanding of computers and the ways they can be used to help in solving problems.

Modern digital computers are essential in a vast range of activities, many of which we now accept as commonplace, but which were impossible to carry out just ten or fifteen years ago. The air transportation industry could not exist, as we know it today, without computers. The computer is used in every stage of the design, construction, and testing of new aircraft. The airlines depend on computers to schedule their planes, to make flight plans, to keep track of passengers reservations, to fly the planes (an autopilot is a special kind of computer), and even to guide the planes on take-off and landing. Other industries have become just about as dependent on computers; for example, nationwide credit card systems would be inoperable without computers.

The exploration of space demands large numbers of computers. A space vehicle must be guided into a very precise orbit. The booster rocket must be carefully controlled in direction and in the length of time it burns. Even slight errors in this control would mean a failure to achieve orbit. To get this precise control, the booster is followed by radar, its position being continually transmitted to a computer which calculates its velocity, acceleration and the changes (if any) that are needed to follow the planned trajectory. Signals from the computer are then transmitted to the booster to control its steering or to cut it off at the right instant. Similarly computers are required for the calculation of course-correction maneuvers, for rendezvous maneuvers, retrorocket firing, and practically every other event occurring in space exploration.

Important uses of computers occur in many other fields. Medicine is making increasing use of computers to aid in the diagnosis of illness and to analyze the progress of a patient's treatment. Computers assist in the editing of literature and even in the automatic setting of type at the printers. Designers (including architects and product designers) are beginning to use computers connected to television screens. They can then display and modify their design as if the screen were a drawing board.

In short, it is hard to think of any field of activity which is not using computers--or, at least, for which potential uses are not being demonstrated in scientific laboratories. No matter what you hope to be, whether you plan to be a homemaker or an astronaut, you are going to use computers. This book will teach you what computers really are and how they can be used--an understanding which will be of tremendous value to you in any vocation.

1-3 Some Technical Aspects of Computers

Two types of electronic computers, analog and digital, are in widespread use. The difference between them lies in the way that each type represents numbers. In an analog computer, numbers are represented by the size of a continuously variable quantity such as the speed of a rotating shaft or an electrical voltage. In a digital computer, numbers are represented by discrete codes to stand for integers (as an example). Analog computers are useful and important but they are not the subject of this book. Other than in the present paragraph, we will always use "computer" to mean digital computer.

We know that a computer has electronic circuits which can somehow store numbers. We don't want to get too deeply involved in how this can be done physically since there are many techniques and an adequate description of any one of them is beyond the scope of this book. We will very briefly describe one of the most common storage devices, the magnetic core.

A magnetic core is a tiny doughnut (perhaps 1/20 inch outside diameter), made of a magnetizable material. Figure 1-1 shows two ways in which a core can be magnetized (that is, clockwise or counter-clockwise).

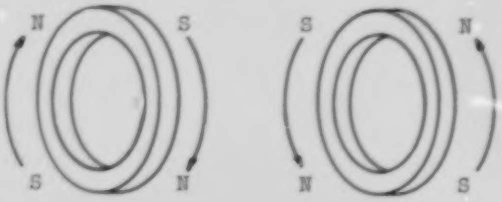


Figure 1-1. Direction of magnetization of cores

You may know that an electrical current can induce magnetism as in an electromagnet. In particular, if a fine wire is wrapped around the core as in Figure 1-2, the direction of magnetization of the core can be controlled by electrical currents in the wire. In fact, the direction of magnetization can be changed very quickly (less than a millionth of a second).

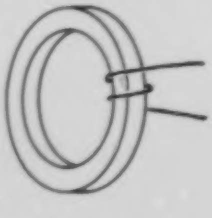


Figure 1-2. A core with wire wrapping

One direction of magnetization can be used to represent the binary digit "0", the other the binary digit "1". Thus, with enough of these wire-wrapped cores we could store, in magnetic form, the binary representation of any number. The catch is that for practical use, we need an awful lot of cores. (A typical small computer may have 100,000 cores, a large computer may have over ten million.) So, the real problem is how a computer selects an individual core to store a binary digit in it or to find out which digit (0 or 1) is already stored in the core.

Typically the magnetic cores are arranged in a plane rectangular array. Imagine a "screen" of very fine wires and a core at each intersection of wires. Each core now has two wires associated with it giving the coordinates of the core in the plane. In reality a wire does not have to be wrapped around a core. It is sufficient for the wires to pass through the hole of the doughnut as is shown schematically in Figure 1-3.

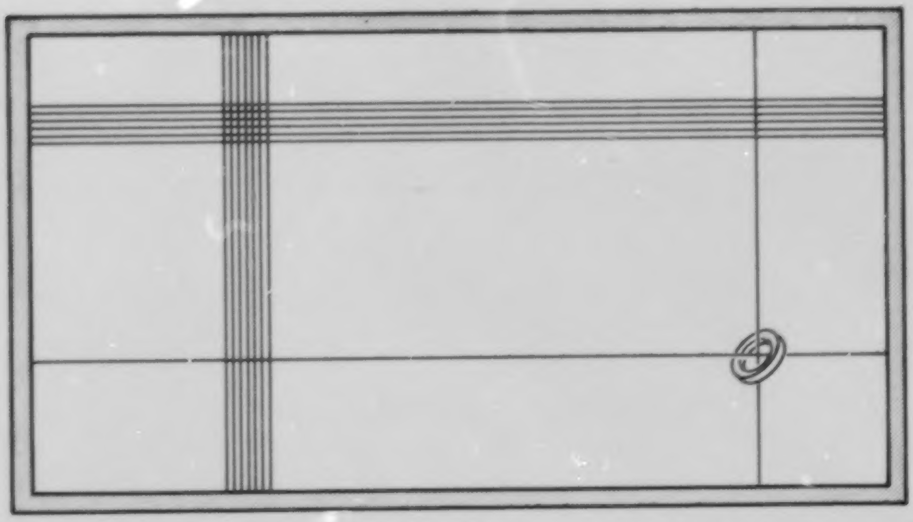


Figure 1-3. An array of magnetic cores (only one core shown)

An individual core from an array which may contain several thousand can now be found by picking just two wires at the same time. Each wire is associated with many cores but there is just one core where the wires cross. The material from which the cores are made has the remarkable property that a critical minimum amount of current is needed to change the direction of magnetization. It behaves like a ball thrown on the sloping roof of a house. If you don't throw hard enough, the ball will come down on the same side as it started. Throw it just hard enough and it will come down on the other side. So, if half (or just a little more) of the critical amount of current is sent along each of the two selected wires in the "screen," only the core at the intersection of the wires can be affected.

To summarize:

(a) Each core can store a binary digit,

(b) an individual core can be selected from among the thousands in a computer, and

(c) the direction of magnetization of a core can be changed in less than a millionth of a second.

Usually the planes of magnetic cores are stacked side by side (or on top of one another). The cores lying in a line perpendicular to the planes (not connected by any wires) are then conveniently treated together as the binary digits (or "bits" from the first two and last two letters of binary digits) of the binary representation of a number. This related group of bits is called a computer word (or simply word, if no confusion results).

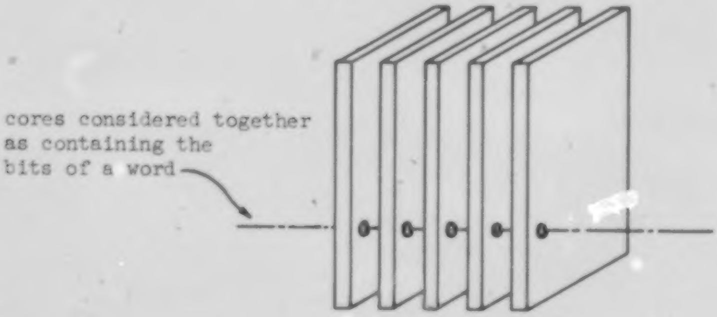


Figure 1-4. A stack of magnetic core planes and the relationship of the bits of a word

Of course, the bits in a word can be used to represent things other than binary numbers. The decimal digits, for example, can be represented by a grouping of four bits for each digit; letters of the alphabet can be represented by a grouping of more bits (at least six are needed for each character if letters and digits are to be represented). Representations of numeric and alphabetic characters will be further discussed in Section 1-4.

The Store

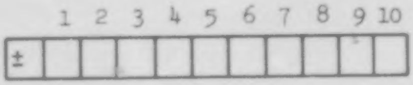
If you have ever shopped in a large supermarket or a department store you already know how helpful it is to have large signs like DRUGS or SUNDRIES to identify the sections in which the drugs and sundries are sold. In general, a means of identifying a location is called an address (like the address of your home or of a post office box).

Here we do not want to discuss supermarkets (or delicatessens) but the section of the computer in which information is stored. This store also uses addresses to identify the different words (made up of bits stored in cores). Most commonly, the integers (including zero) are used for addresses in a computer store. The word memory is often used in place of store. Some people have preferences for one word or the other but we will use them interchangeably.

From the description of how magnetic cores work, it should be clear that only one thing can be stored in a word at one time. Therefore, when new information is placed in a word of store, whatever was there before is destroyed. This is called destructive read-in.

We have not told you what happens when information stored in magnetic cores is requested. Actually, computers are built so that a copy of what is stored in a word is made in response to a request for that information. The original information remains in the store. This is called non-destructive read-out.

A hypothetical computer called SAMOS (which doesn't stand for anything special) is described in Appendix A. Each model of computer is different in detail but SAMOS is intended to be representative in its overall plan. The store of SAMOS consists of 10,000 words each of which can contain a sign (+ or -) and ten letters or digits so that a SAMOS word has groupings of bits (from left to right).



Structure of a SAMOS word

It is convenient to think of the computer store as an arrangement of boxes (or pigeonholes, such as those used for sorting mail). Each box corresponds to a computer word and can contain just ten characters at a time. The boxes are identified by the integers 0000, 0001, ..., 9999 which are the addresses of SAMOS.

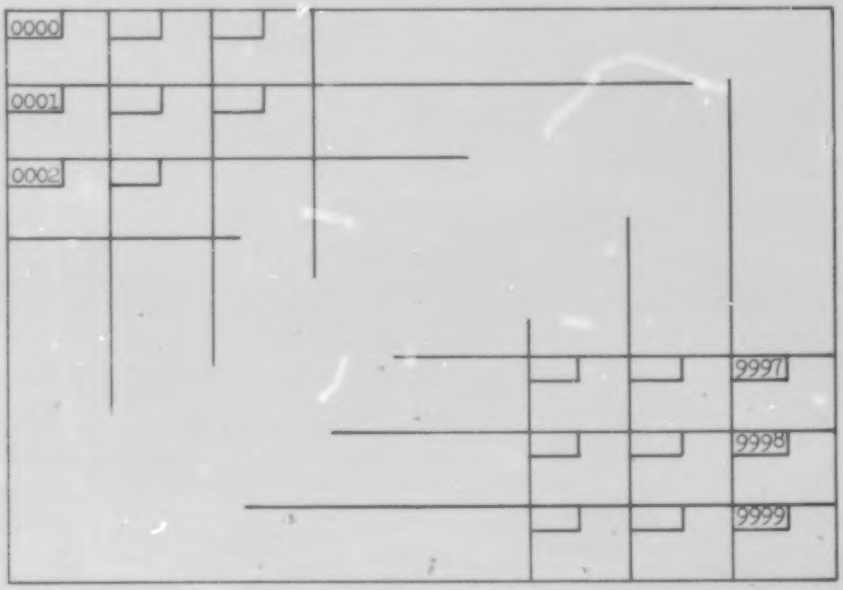


Figure 1-5. The store of SAMOS

A SAMOS instruction is ten characters in length and it always has a + sign at the beginning. Thus, one instruction fits in one word of the store (remember the stored program concept from Section 1-2).

Numbering the ten positions of a word from left to right, every SAMOS instruction has the following form:

Positions	Meaning
Sign	No meaning, assumed always + .
1,2,3	A code for the operation to be performed. For example, ADD for addition, MPY for multiplication, etc.
4,5,6	We will always assume zero in these positions although Appendix A does make use of them.
7,8,9,10	A four-digit number, giving the address of a SAMOS word.

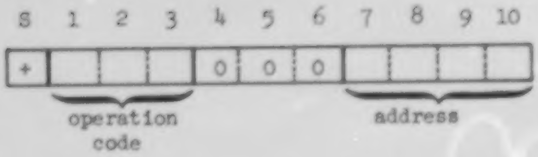

S 1 2 3 4 5 6 7 8 9 10
+
0 0 0
operation code
address

Figure 1-6. Structure of a SAMOS instruction

The Arithmetic Unit

The arithmetic unit is connected to the store of a computer so that it can receive a copy of information from any word of store or transmit the result of a calculation it performs to replace whatever may be in any specified word of the store.

Arithmetic units contain the electronic circuits to perform arithmetic operations on data. They may also contain one or more special storage devices called accumulators. (SAMOS contains one accumulator, see Figure 1-7.) An accumulator is used to hold temporarily the result of an arithmetic operation.

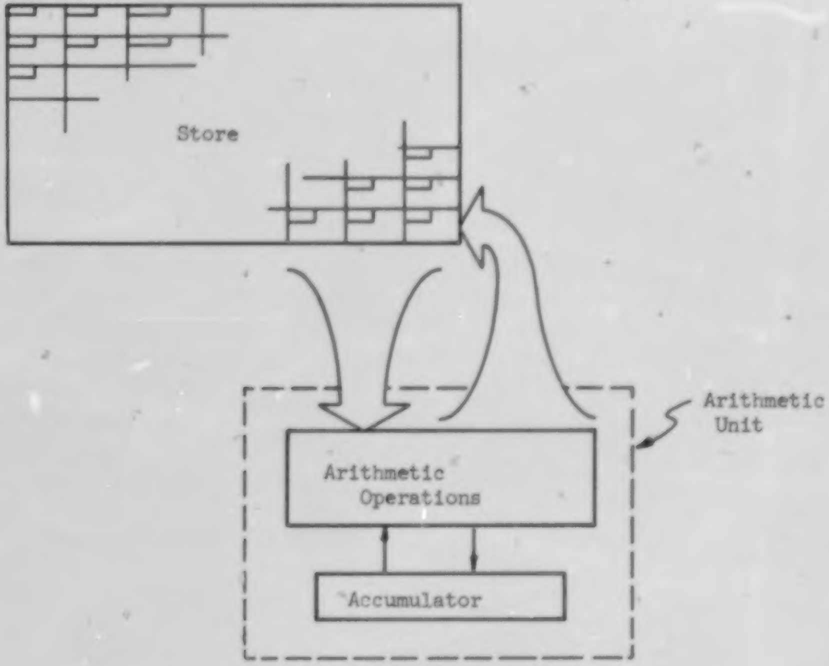

Store
Arithmetic Unit
Arithmetic Operations
Accumulator

Figure 1-7. Relation of Arithmetic Unit and Store in SAMOS

The arithmetic operations that SAMOS can do include replacement of the content of the accumulator by the content of a specified word of the store, and vice versa (generally called "loading" the accumulator and "storing" the accumulator). It can perform addition to the accumulator, subtraction from the accumulator, multiplication by the accumulator, and division into the accumulator. In each case the result remains in the accumulator. All the more complicated arithmetic operations (like a square root or the sine of an angle) are built up out of these basic operations.

Each operation has an operation code which the computer has been designed to interpret. For example, LDA is the SAMOS operation code meaning, replace what is in the accumulator by a copy of what is in the word, the address of which is given in the instruction. This meaning is usually abbreviated to "load the accumulator."

If a person speaks precisely about what is happening in a computer, he tends to become involved in such complex (and hard to unravel) sentences as in the last paragraph. For this reason, destructive read-in and non-destructive read-out are generally assumed in speech and writing. Moreover, a distinction must often be made between the address of a word (a post office box number) and the information contained in the word (a letter in the post office box). If L stands for the address of an arbitrary word in the store, one popular way to indicate the content of L is with parentheses. That is, (L) represents the content of address L. With this notation, the arithmetic operation codes are defined as in Figure 1-8, where L is the address given in the instruction and "←" is to be read as "is replaced by."

Operation Code (positions 1,2,3)	Meaning	Result
LDA	Load the accumulator	(ACC) ← (L)
STØ	Store the accumulator	(L) ← (ACC)
ADD	Add to the accumulator	(ACC) ← (ACC) + (L)
SUB	Subtract from the accumulator	(ACC) ← (ACC) - (L)
MPY	Multiply by the accumulator	(ACC) ← (ACC) × (L)
DIV	Divide into the accumulator	(ACC) ← (ACC)/(L)

Figure 1-8. Basic SAMOS arithmetic operations

Of course, to get anything done, a series of instructions has to be executed.

An historic example

Suppose we have been told that numbers are stored in addresses 1066, 1492 and 1776 in SAMOS and that to solve some problem we should multiply (1066) by (1492) and to that product add (1776). We are also told to store the result of the calculation in address 1965.

In accordance with what we have been told we would write:

Instructions		Comments (or meaning)
pos.1-3	pos.7-10	
LDA	1066	load the accumulator with (1066)
MPY	1492	multiply (1066) now in accumulator by (1492)
ADD	1776	add (1776) to the product in the accumulator
STØ	1965	store the result in 1965

Notice that we are not, as yet, saying where the instructions are. We merely assume that the instructions will be executed from top to bottom in the order they appear.

The Control Unit

The control unit is the part of the computer designed to determine which instruction is to be done next, to decipher the operation indicated by that instruction code, and to establish the connections between electronic circuits carrying out the operation (recall the stored program concept). The control unit of SAMOS can be thought of as containing three specialized storage devices; the instruction counter, the operation register, and the address register.

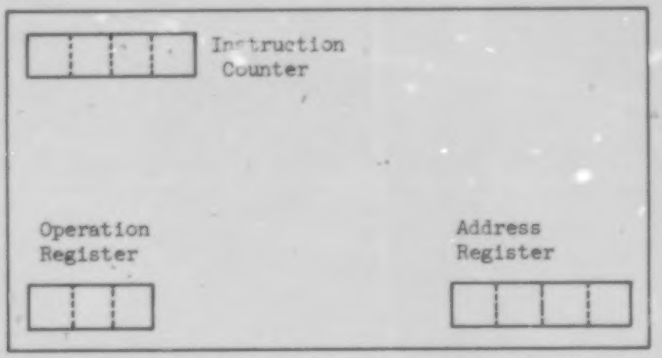


Figure 1-9. Registers in the control unit

To begin with, the instruction counter is set to show where a program is to begin, that is, it contains the address of the first instruction to be performed. Thereafter, every time an instruction is used, the instruction counter goes up another notch. The instruction counter contains the address of the next instruction to be used. As soon as circuits have been set up to bring a copy of that next instruction to the control unit, the instruction counter (by automatically having one added to its content) "counts" to the next instruction to be done. You can see that instructions will be used in the sequence in which they appear in the words of the store. Therefore, SAMOS is called a sequential computer.

A copy of the operation part (Positions 1, 2, and 3) of the instruction to be executed is placed in the operation register and a copy of the address part (Positions 7, 8, 9, and 10) is placed in the address register. What is in the operation register is used to connect electronic circuits in the arithmetic unit (or wherever else needed) in preparation for doing the particular operation specified. What is in the address register is used to make a connection between the particular word in the store and the arithmetic unit. Then the operation is executed.

After execution of an instruction, the instruction counter is again used to bring the next instruction to the control unit and the cycle is repeated. It is worth pointing out that if SAMOS can execute 100,000 instructions per second (a reasonable rate), all the instructions in its memory at one time would be used up in a tenth of a second unless some way were found to break the sequencing. Moreover, even ten thousand instructions could not do a simple thing like finding the absolute value of a number without some way to select the sequence of instructions to be used.

The cycle of using instruction after instruction in sequence can be broken if the address in the instruction counter is replaced by a different address. SAMOS has several instructions (generally called branch instructions) to effect this replacement. We will mention just three such instructions in Figure 1-10 (refer to Appendix A for more detail).

Operation Code	Meaning	Result
BRU	Execute next the instruction at L.	$(IC) \leftarrow L$
HLT	Halt the machine. If the start button[†] is pressed, execute the instruction at L.	Halt and $(IC) \leftarrow L$
BMI	If the sign of the accumulator is negative, execute the instruction at L. Otherwise BMI has no effect.	if $(ACC) < 0$, $(IC) \leftarrow L$ otherwise $(IC) \leftarrow (IC)+1$

Figure 1-10. SAMOS branch instructions

An absolute example

In this example we will store in address 1234, the absolute value of the number A, which is in address 4321. Suppose the instruction counter starts at 1001, the address of the first instruction of the example.

Inst. Location	Instruction		Comments
1001	LDA	4321	load accumulator with the number, A
1002	BMI	1004	is A negative?
1003	BRU	1006	A is already ≥ 0 if you reach this instruction
1004	SUB	4321	this puts a zero in the accumulator
1005	SUB	4321	-A in accumulator
1006	STO	1234	store $\|A\|$

The first instruction brings the number A to the accumulator and the instruction counter advances to 1002. The sign of the accumulator is then tested by the BMI instruction. If the sign of A is negative, something will have to be done to change it. In this case, BMI will cause 1004 to be inserted into the instruction counter; otherwise the instruction counter advances to 1003. If the instruction counter reaches 1003, we know that A is non-negative and its value is to be stored in 1234. Therefore, 1003 contains an unconditional branch instruction to 1006 which contains the instruction to store the contents of the accumulator in 1234. If the instruction counter reaches 1004, we know that A is negative. The instruction in 1004 subtracts A from itself--a convenient way to put zero in the accumulator--and the instruction counter advances to 1005. There the negative of A is placed in the accumulator, the instruction counter advancing to 1006 which is the instruction to store the contents of the accumulator in 1234. You can trace the sequence of instructions performed in either case as follows:

[†] SAMOS, like all computers, has buttons so that people can start and stop (and otherwise control) its operation.

Instructions performed

If $A < 0$	If $A \geq 0$
1001	1001
1002	1002
1004	1003
1005	1006
1006	

1-4 Numbers and Other Characters

In developing computer programs we will need to know how the data dealt with in problems is represented. The SAMOS computer (see Appendix A) treats every number as though it were an integer. When two numbers are multiplied they are treated as integers. When one number is divided by another, the quotient is treated as an integer, the remainder (i.e., the fractional part of the quotient) being discarded. This is called integer division. Examples of integer division are given in Chapter 2. How then can SAMOS use numbers or get answers that are not integers? The easiest answer is to require the person writing a SAMOS program to multiply each number by factors of ten so that the machine can perform operations (e.g. division) without discarding desired digits. This amounts to positioning (or shifting) the numbers in the computer word so that the machine, in doing its integer arithmetic, is "fooled" into keeping digits that can be reinterpreted as being in the fractional part of a number. If two numbers are to be added or subtracted the programmer should be certain that each contains the same number of digits interpretable as being in the fractional part. This is the easiest answer but it is a complicated task for the programmer to do.

Notice that the main difficulty is in keeping track of where the decimal point is in a number. Long ago, mathematicians, scientists, and engineers developed a convenient scheme called "scientific notation" for positioning the decimal point. The method is to write every number with a decimal point after the first non-zero digit; then each number is multiplied by a power of ten sufficient to position the decimal point properly.

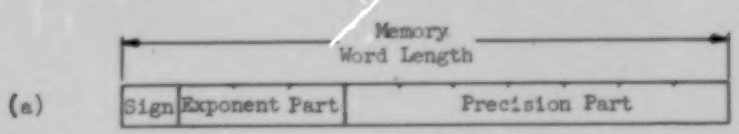
For example:

Usual notation		Scientific notation
3.1415926	=	3.1415926×10^0
-273.14	=	-2.7314×10^2
.0008761	=	8.761×10^{-4}

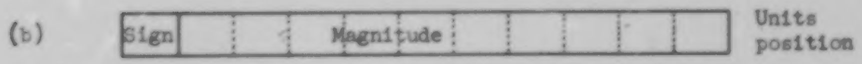
Many computers are constructed so that they can use numbers in a form similar to scientific notation. Numbers that are represented inside a computer by making use of the exponent idea are said to be represented in floating point form. A computer constructed to use numbers in floating point form has different instructions for adding two numbers in floating point form and for adding two numbers treated as integers. Even if a computer is not constructed to use this (floating point) representation, standard programs for each type of computer (from a "library of programs") can be used to perform arithmetic with numbers in a floating point form.

Use of floating point numbers, that is, numbers in floating point form, makes it much easier to do arithmetic since the computer keeps track of the position of the decimal point for you.

Our comparison of integer form and floating point form can be made more explicit by reference to Figure 1-11 which shows these two forms for representing numbers in a given word of memory.



floating point form



integer form

Figure 1-11. Illustrating two forms for internally representing numbers:

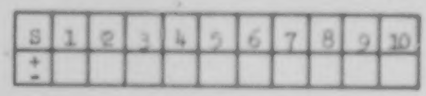
(a) used for "real" numbers including integers

(b) used for integers only

Notice that numbers with fractional parts cannot be represented in the integer form as long as the units position is agreed to be fixed at the right end of the word. On the other hand an integer can be represented in either form.

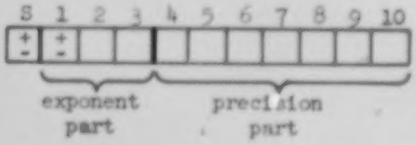
To represent an integer, the integer form seems like the natural choice, because this form has a far simpler structure. Circuitry for performing arithmetic on numbers which are coded in an integer form is less complicated. It works faster, and is cheaper to build. On the other hand it is worth noting that integers which are too large to fit in a given word length in the integer form can be adequately approximated in a floating point form which uses the same word length.

To explain this point, let us imagine we have a storage word which holds decimal digits. Assume the word length is sufficient to hold eleven decimal digits, but that the left most of these is preempted for storing the sign of the number. We'll show it in the figures below as $\boxed{\pm}$ to indicate that either sign may be present.



A word containing an integer

In the integer form, the largest integer which can be represented is +9999999999. Now one way to employ this memory word in a floating point form might be to treat positions 1, 2 and 3 as the exponent part, using position 1 as a sign for the exponent, and to treat positions 4, 5, ..., 10 as a seven-position precision part.



A word containing a floating point number

Let us further agree that the precision part always represents a number lying between .1000000 and .9999999 (i.e., to seven digit precision).

With these rules the numeric examples mentioned earlier are shown in Figure 1-12, as they would appear internally.

EXTERNAL		INTERNAL
Number	Computer notation	Floating point form
3.1415926	$.31415926 \times 10^1$	+ + 0 1 3 1 4 1 5 9 2
-273.14	$-.27314 \times 10^3$	- + 0 3 2 7 3 1 4 0 0
.0008761	$.8761 \times 10^{-3}$	+ - 0 3 8 7 6 1 0 0 0

Figure 1-12. External and internal representations of numbers

You can see that the largest number which can be coded in this floating point form, i.e.,

S	1	2	3	4	5	6	6	8	9	10
+	+	9	9	9	9	9	9	9	9	9

would represent

$$+.9999999 \times 10^{99}$$

which is far larger (although with not as many digits of precision) than can be represented in the integer form.

It would seem that any calculation using real numbers could be done by computers using the floating point representation. This is not really true (we will come back to this point, particularly in Chapter 6) because the part of a floating point number following the decimal point (called the precision part) is of limited length. Nevertheless, in computer jargon, we speak of "integer" numbers and "real" numbers in algorithms even though not all of the integers and not all real numbers can be represented in a computer.

The examples of the last few pages make use of decimal numbers because these are the numbers we are accustomed to. You must not think that floating point forms are limited to decimal numbers. Many computers, including the most powerful, represent numbers in binary form, and by using the same ideas outlined above, perform arithmetic with binary floating point numbers.

Computers that perform arithmetic with decimal numbers must somehow store the decimal digits in magnetic cores each of which can store a bit. How can this be? Are you familiar with any way to group bits so as to represent the decimal digits? There are literally hundreds of ways to accomplish such a representation. The most obvious way is to use the binary forms of the decimal digits directly.

Decimal Digit	Binary Form	Decimal Digit	Binary Form
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Figure 1-13. Binary forms of decimal digits

Since each decimal digit is coded into binary form, this representation is called binary coded decimal (or BCD, for short). The code for each decimal digit takes the place of that digit so that, for example,

365 is coded as 0011 0110 0101

and 1965 is coded as 0001 1001 0110 0101

Representing all kinds of characters

In general, n bits can be used to represent 2^n different things. Thus 3 bits could be used to represent $2^3 = 8$ things--not enough for the ten decimal digits. Four bits give $2^4 = 16$ combinations, enough for the decimal digits with some left over. If we want to represent not only the decimal digits but also the 26 letters of the alphabet, we would need at least $26 + 10 = 36$ combinations. Five bits gives $2^5 = 32$ combinations which are not enough. Six bits are required to provide $2^6 = 64$ combinations with plenty of combinations left over for other characters such as $, =, ×, blank space, etc.

A popular type of coding places two extra bits to the left of the four bit binary forms of Figure 1-13. This type of coding is displayed in Figure 1-14.

Character	Code	Character	Code	Character	Code	Character	Code
0	00:0000						
1	00:0001	A	01:0001	J	10:0001		
2	00:0010	B	01:0010	K	10:0010	S	11:0010
3	00:0011	C	01:0011	L	10:0011	T	11:0011
4	00:0100	D	01:0100	M	10:0100	U	11:0100
5	00:0101	E	01:0101	N	10:0101	V	11:0101
6	00:0110	F	01:0110	Ø	10:0110	W	11:0110
7	00:0111	G	01:0111	P	10:0111	X	11:0111
8	00:1000	H	01:1000	Q	10:1000	Y	11:1000
9	00:1001	I	01:1001	R	10:1001	Z	11:1001

Figure 1-14. Six bit code frequently used to represent both digits and alphabetic characters

The Input Problem

Numbers, letters and other characters of the data must be introduced (read) into the computer through some input device. If there is a "first law of computer input", it probably goes something like: "We always want to make use of more characters than can be recognized by the available reading device." One will quickly discover, in fact, that we would like even more characters than can be encoded with six bits. Typically, restrictions on the set of recognizable characters result either from the extra expense of building a device that can recognize more characters than seem to be really necessary, or from the fact that the traditional set of characters for common input devices antedates computers.

In the banks, account numbers are now commonly printed at the bottom of checks. In printing, an ink is used containing a ferrous compound which can be magnetized. When magnetized, the printing characters can induce a current in a reading device and the individual, "funny" shapes of the printed characters induce distinguishable shapes of current. This system, "magnetic ink character recognition" or MICR, was particularly developed for the use of banks where numeric coding is adequate. To avoid unnecessary expense, the character set has been limited to digits and a few extra characters used as separators.

The electric typewriter is often used as an input and output device for computers directly or with punched paper tape as an intermediate step. This device provides the possibility of machine recognition of all the characters (upper and lower case) found on a typewriter. The limitation therefore is the character set normally found on a typewriter. Unfortunately, because of expense, all of the available characters are not always used.

Typewriters, teletypes, and other machines with similar keyboards are coming into common use to communicate with computers at a distance. All this amounts to is making a phone call to the computer. Since dozens of typewriters can now be connected to a single computer, the set of characters that can be used for input is more often becoming the ordinary typewriter set.

Punched cards

The keypunch is a machine that has developed with the use of punched cards and punched card tabulating systems for more than 60 years. In tabulating systems it proved to be uneconomical to use more than the digits and capital letters together with a few so-called "special" characters. Here, the special characters are even a "recent" addition of the past thirty years.

When a key is struck on a keypunch, a pattern of holes is produced in a column of a card. These holes can subsequently be identified by a card reading machine. The codes used in key punches in most common use today are designed in this way: To produce the plus or minus sign or one of the ten digits (0, 1, ..., 9), a single hole is punched in one of 12 rows of the card. To represent the capital letters (A ... Z), they produce a combination of two holes (see Figure 1-15). Note that of 12 × 11/2 = 66 possible two-hole combinations, only 26 are used (really 27 are used because the special character "/" has a two-hole encoding). Beyond this there are eight special characters encoded as three holes punched in a column. (How many three-hole combinations are possible?)

The keypunch has been in common use to transmit information to a computer because of the relative permanence of the cards (they can be used over and over), and because the individual cards can be readily reordered or replaced. However, the usual keypunch encodes only 48 characters (including no punch for a blank space). Fortunately, equipment manufacturers have paid considerable attention to the problem of providing a larger character set at reasonable cost. Currently several devices (typewriters, keypunches, and fancier machines) which encode "extended character sets" are being marketed. In the absence of these newer machines unavailable characters are usually coded as combinations of those available (e.g., $A conceivably could be used to imply lower case a).

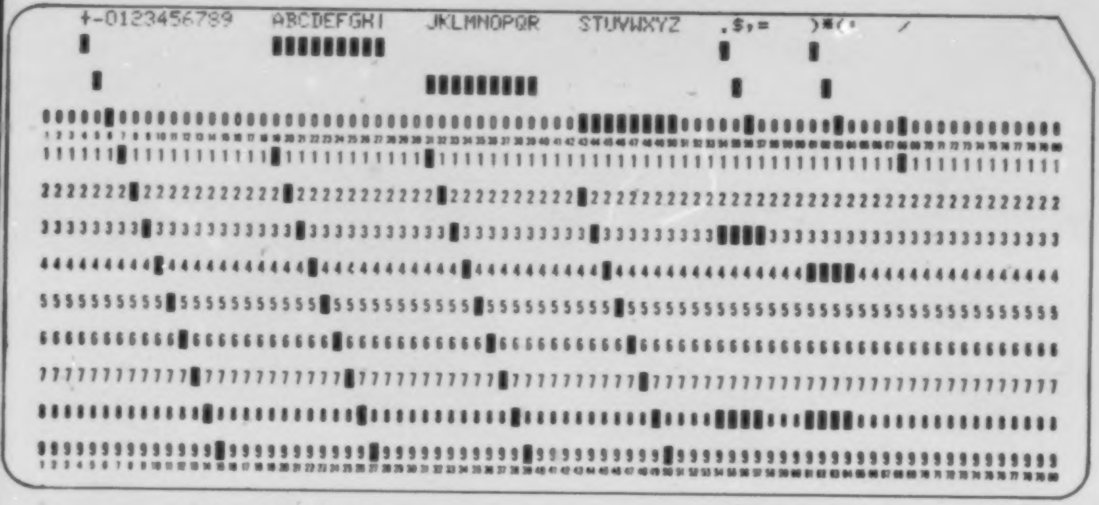


Figure 1-15. Punch card codes for a typical character set

This coding scheme is also referred to as Hollerith Code in honor of Herman Hollerith, a late nineteenth century statistician who developed the first punch card sorting and tabulating equipment.

1-5 Algorithms

Now that we have learned something of how a computer works, let's find out how to prepare a problem for computation. One of the striking things about computers is that all they can do at one time is one fairly small step, like add to the accumulator, or store the accumulator. To be able to do anything more complicated the computer must execute a sequence of instructions. A sequence of instructions for a computer is called a computer program.

A more general name for a sequence of instructions to solve a problem, whether with a computer or not, is "algorithm". An algorithm actually has characteristics that a computer program may not have--although most useful computer programs do. Any arbitrary sequence of instructions could be a computer program but an algorithm must, in addition, give an answer to a problem within a finite number of steps.

More formally, an algorithm is any unambiguous plan telling how to carry out a process in a finite number of steps. You should be able to think of lots of examples of algorithms; some examples could be the instructions for assembly of a model airplane, or the score for a piece of music. Each of these examples is a set of instructions designed to produce a specific result and each comes to an end.

A characteristic of algorithms that has already been mentioned but must be emphasized is that the step-by-step plan must be unambiguous. We can not tell a computer to "either add or subtract." Rather, we must say--"if specific, detailed conditions are satisfied, then add; if these specific conditions are not met, then subtract." There can be no room for doubt as to the meaning of an algorithm.

Algorithms that are useful with computers frequently have several other characteristics. First among these is generality. For example, it is not very interesting (or useful) to know that the greatest common divisor of 12 and 36 is 12. It is far more useful to know a step-by-step way of finding the greatest common divisor of any two integers, a and b (Euclid's algorithm). The same is true of any useful algorithm. For example, an algorithm to find the solution set for a quadratic equation,

$$ax^2 + bx + c = 0$$

should produce answers (or tell us there are no answers) for any values of a, b and c (even for the degenerate case in which a might be zero).

A second common characteristic of useful algorithms is repetition. Instructions for assembly of a model airplane often say "repeat Steps 7 to 12 for the right wing as for the left wing." In a musical composition, special symbols are used to tell a performer to repeat a part of the piece. In the same way, we will find it extremely useful to repeat series of steps in a computing algorithm. One reason computers are so useful is that useful algorithms do depend heavily on repetition and the computer will repeat the same steps tirelessly and without complaint.

Using Algorithms in Solving Problems

Sometimes we connect the word "problem" with a question on an examination or a homework assignment. More generally, "problem" means any situation in which there is a difference between what one has and what one wants. If you don't have a date for a dance, and you want a date, you have a problem. If you have to plan a menu for the school cafeteria and you want to include nourishing foods that people like without costing too much, you have a problem.

Problems can be separated into real-life problems like those above and composed problems like those in most textbooks. Algorithms are important in helping to solve both kinds of problems but real-life problems like getting a date or planning a menu often have a very large number of possible choices to be made in finding a solution, making them hard to analyze. You can probably write down a series of steps you would go through in getting a date but if someone else could interpret your instructions in a different way, or if your instructions wouldn't get a date every time, you have not written an algorithm!

Besides the large number of choices to be made in finding a solution, real-life problems are hard to discuss precisely because we tend to use our native language, English, and this has its own built-in ambiguities. So to discuss algorithms we must consider all alternatives and express our instructions in such a way that they cannot be misunderstood. To see what can be done, consider a problem with which you may already be familiar.

Example 1

Suppose you are given eight balls all of which look alike, but you are told that one ball is heavier than the others which are identical. Equipped only with balance, identify the heavy ball in no more than three weighings.[†]

First, we will introduce symbols to avoid the ambiguities of English. Label the balls A, B, C, D, E, F, G, and H. Let the weights of the balls correspondingly be a, b, c, d, e, f, g, and h. Now we can write a series of steps to solve the problem.

```
1.  If $a + b + c + d < e + f + g + h$ jump to Step 9.
2.  If $a + b < c + d$ jump to Step 6.
3.  If $a < b$ jump to Step 5.
4.  The heavy ball is A. End of calculation.
5.  The heavy ball is B. End of calculation.
6.  If $c < d$ jump to Step 8.
7.  The heavy ball is C. End of calculation.
8.  The heavy ball is D. End of calculation.
9.  If $e + f < g + h$ jump to Step 13.
10. If $e < f$ jump to Step 12.
11. The heavy ball is E. End of calculation.
12. The heavy ball is F. End of calculation.
13. If $g < h$ jump to Step 15.
14. The heavy ball is G. End of calculation.
15. The heavy ball is H. End of calculation.
```

Studying these fifteen steps you will find that the problem is solved in all cases and misinterpretation of the steps would be difficult if not impossible. Therefore this is an algorithm. Still, the fifteen steps seem to be a complicated answer to what appears to be a simple problem. Moreover, the fact that only three weighings are required in any particular case is not obvious since there are seven possible weighings listed in the algorithm. It would be interesting to have a means to clearly distinguish between the seven weighings of the algorithm and the three sequential weighings done when the algorithm is executed.

There is an elegant diagrammatic way to display the solution to this problem. First, use a colon (:) to represent the comparison operation of weighing on the scales so that, for example $a + b : c + d$ means place balls A and B on the left pan of the scales and compare with balls C and D placed on the right pan. Enclose each comparison (or other step) in a box and adopt the convention that if the weight on the left of the comparison is heavier we will "leave" the box on the left; if the weight on the right of the comparison is heavier we will "leave" the box on the right. With these conventions, a diagram of the earlier fifteen step algorithm is shown in Figure 1-16.

[†] More complicated problems of this type can be stated and easily solved with the methods discussed here.

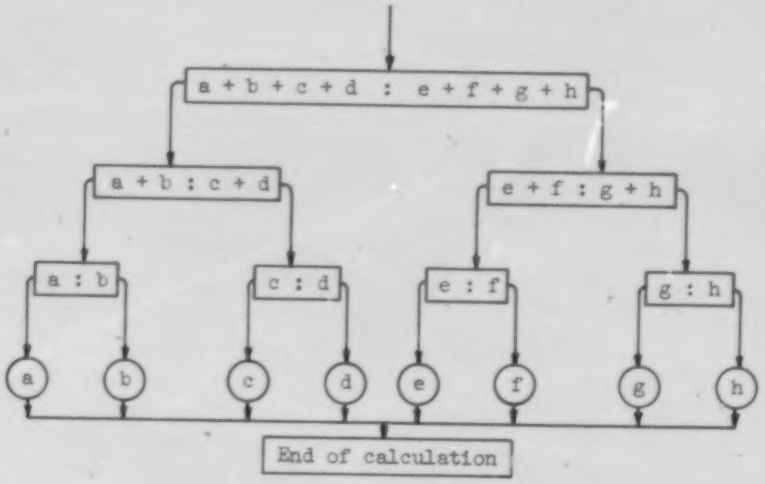


Figure 1-16. Diagram of algorithm to find a single heavy ball from a set of eight balls

Here we have a clear display of the three sequential weighings needed to select the heavy ball and the plan of the process stands out. This example alone, should be enough motivation to further study diagrammatic representations of algorithms. In the next several chapters we will develop a diagrammatic language (called the flow chart language) for representing algorithms.

Just for the fun of it you may want to try to find diagrams for the following related problems:

1. Suppose you are given eight seemingly identical balls and you are told that one ball is different in weight (either heavier or lighter). Identify the ball and whether it is heavier or lighter in three weighings.
2. Suppose you are given twelve seemingly identical balls and you are told that one ball is heavier than the others, which are the same weight. Identify the heavy ball in three weighings.

3. Suppose you are given fourteen seemingly identical balls and you are told that one ball is heavier than the others, which are the same weight. Identify the heavy ball in three weighings.

4. Suppose you are given twelve seemingly identical balls and you are told that one ball is different in weight (either heavier or lighter). Identify the ball and whether it is heavier or lighter in three weighings.

Example 2

For a second example to illustrate the generality desired in an algorithm let us turn to a more mathematical problem; that of finding the real solution set of the quadratic equation

$$ax^2 + bx + c = 0$$

for any set of real numbers a, b, and c. You know that (as long as a is not zero):

if $b^2 - 4ac < 0$ there is no real solution,

if $b^2 - 4ac = 0$ there is one real solution given by $x = -\frac{b}{2a}$,

if $b^2 - 4ac > 0$ there are two real solutions which are:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad \text{and} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} .$$

Since we want an algorithm so general that values of a, b, and c could be numbers radioed from Mars, we will look for pathological cases. What if $a = 0$? What if $a = b = 0$? If $a = 0$ and $b \neq 0$, the equation degenerates into a linear equation with one real solution,

$$x = -\frac{c}{b} .$$

If $a = 0$ and $b = 0$, then if $c = 0$, any real value of x can satisfy the quadratic equation, or if $c \neq 0$ there is a contradiction and the three numbers cannot be coefficients of a quadratic equation. In either of these cases we would be justified in saying that there is no interesting solution. Are there any other special cases? If $a \neq 0$ and $b = 0$, the quadratic formula applies but we can make the calculation shorter by recognizing this case separately, then $x_1 = \sqrt{\frac{-c}{a}}$ and $x_2 = -\sqrt{\frac{-c}{a}}$ provided c/a is negative. These special cases are summarized in Figure 1-17.

	$a = 0$	$a \neq 0$
$b = 0$	no interesting solutions	$x_1 = \sqrt{\frac{-c}{a}}$ and $x_2 = -\sqrt{\frac{-c}{a}}$
$b \neq 0$	$x = -\frac{c}{b}$	usual rules apply

Figure 1-17. Special cases in solution of quadratic equations

Now let's write down a list of steps to do this calculation.

1. If $a \neq 0$ jump to Step 5,
2. If $b \neq 0$ jump to Step 4,
3. There are no interesting solutions. End of calculation.
4. One solution, $x = -c/b$. End of calculation.
5. If $b \neq 0$ jump to Step 8,
6. If $c/a > 0$ jump to Step 11,
7. Two solutions, $x_1 = \sqrt{-c/a}$ and $x_2 = -\sqrt{-c/a}$. End of calculation.
8. Calculate discriminant = $b^2 - 4ac$,
9. If discriminant > 0 jump to Step 13,
10. If discriminant $= 0$ jump to Step 12,
11. There are no real solutions. End of calculation.
12. One solution, $x = -b/2a$. End of calculation.
13. Two solutions, $x_1 = (-b + \sqrt{\text{discriminant}})/2a$ and $x_2 = (-b - \sqrt{\text{discriminant}})/2a$. End of calculation.

Read this carefully. Are any situations not covered? Can you misinterpret these instructions? This is an algorithm but it is not, in fact, the whole story concerning the solution of a quadratic equation. In Chapter 6 we will return to the quadratic equation to discuss other difficulties that this algorithm could encounter and to discover how this algorithm can be repaired to account for those difficulties.

To help find a diagram for this algorithm we will adopt the convention that lines leaving a box in which a comparison (with =, ≠, >, etc.) takes place will be labelled T and F for true and false. Then Figure 1-18 is a diagram for the quadratic equation algorithm.

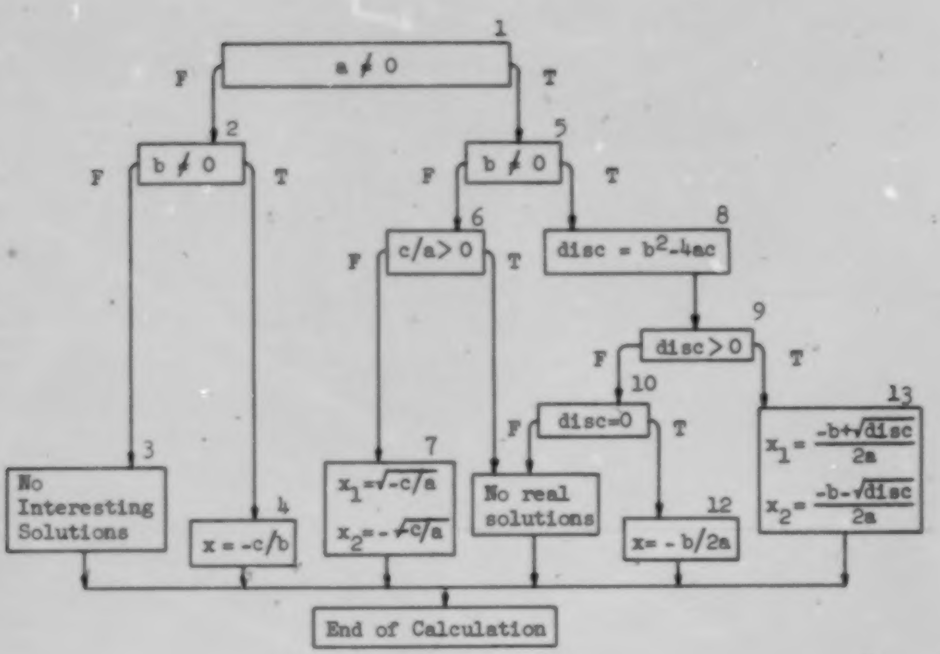

1
a ≠ 0
F
T
2
b ≠ 0
F
T
5
b ≠ 0
F
T
6
c/a > 0
F
T
8
disc = b²-4ac
9
disc > 0
F
T
10
disc=0
F
T
13
x₁ = (-b+√disc)/2a
x₂ = (-b-√disc)/2a
3
No Interesting Solutions
4
x = -c/b
7
x₁=√-c/a
x₂=-√c/a
No real solutions
12
x= - b/2a
End of Calculation

Figure 1-18. Diagram of an algorithm to solve a quadratic equation

The examples given in this section show that it is vital to use unambiguous language forms in describing an algorithm. Problems that lend themselves to such unambiguous language (generally mathematical and logical problems) are those for which we can most easily express algorithms. On the other hand, real-life problems like planning a menu or getting a date for a dance, can be broken down into step-by-step processes. When we can express these processes unambiguously (which we are not used to doing) and when the processes account for all possible situations, the solutions to real-life problems can be given as algorithms.

A "real-life" problem which shows the usefulness of algorithms particularly well and which we challenge you to solve on your next free weekend is The Concentration Camp Problem:

Two men are confined to a cell in a concentration camp. Each day they are given one loaf of bread. They then face the problem of how to divide the bread so each is satisfied that he has received his share. The classical solution is that one divides the loaf and the other takes first choice.

Now suppose that a third prisoner is put in the cell. How then are they to divide the loaf? Your solution must be such that if any prisoner is dissatisfied, it will be in consequence of his own greed or poor judgement. It must be proof against the collusion or illogical behavior of others. The best solution provides an algorithm which is easily extended to any number of prisoners.

1-6 Comments on Language

Language is a means of expressing and communicating our ideas. In computing we want to be able to communicate not only with people but also with computers. To communicate with people we normally use the "natural language" we learned as children. Still, in specialized topics, people have always found it useful to devise specialized jargons and languages.

To communicate with a computer we have to be able to express algorithms unambiguously in a form the computer can understand. We have already had a taste (in Section 1-3) of what is involved in writing instructions for a computer in "machine language." Since each model of computer has differences in design and these show up as differences in each machine language, a "Tower of Babel" situation exists in which a program prepared in machine language for one computer cannot be used by another computer.

In Section 1-2, we remarked that a significant recent development has been the exploitation of procedural languages, specifically designed for the expression of algorithms in a form computers can understand, essentially independent of a particular model of computer.

The flow chart language developed in this book is a formalization of the diagrammatic way we displayed algorithms in Section 1-5. You will discover that the flow chart language helps us to develop, display, and discuss algorithms in an unambiguous way.

In light of these remarks, the task of using a computer to help in solving a problem can be separated into three distinct steps: that of reducing our problem to a sequence of elementary steps; that of "formalization" or converting to a formal language; that of transforming from this formal language to machine language. Each of these steps is a translation from one form, or language, to another. Consequently, a diagram of the translation process would look like Figure 1-19.

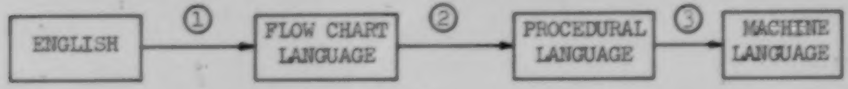


Figure 1-19. Actual Translation Process

The first translation involves making a diagram called a "flow chart" for the problem. Each box of the flow chart corresponds to a task to be performed in the order shown by arrows between the boxes (as in Figures 1-16 and 1-18). What is written inside each box is supposed to explain the task but need not be a detailed description. In fact, relatively complicated problems are often solved by subdividing tasks into simpler, more easily understood subtasks. At each stage the problem solution is described by a flow chart. Explanations of tasks inside the boxes become more specific and more formal as we approach a description of the final solution. This book is chiefly concerned with the flow chart language and its use as an aid in discovering problem solving processes.

The second translation expresses the flow chart in a formal language which is in general use such as FORTRAN or ALGOL. It is the purpose of the FORTRAN and ALGOL language supplements accompanying this text to teach you this second translation process.

The third translation is from one completely formal language to another such language. You can imagine that given two languages which have strict rules of expression and are free from ambiguities, translation from one to the other ought to be performable by machines. This is in fact the case. If you have, say, an ALGOL program for your problem, the last translation is made by feeding your program into the computer along with a "compiler" (or translating) program to produce a translation of your program into machine language.

The purpose of the discussions just completed was to exhibit the relationship of our activities in this book to the general problem of feeding a mathematical problem into a machine. With these words we are ready to proceed with the main business of the rest of this book--that of constructing flow charts.

The material in the language supplement is arranged roughly in parallel with the subject matter covered in this text. For Chapter 2 it is preferable to read the chapter in its entirety before studying the corresponding chapter in your language manual. For subsequent chapters it will be feasible, unless otherwise indicated, to read a section of the language supplement just after you have read the correspondingly numbered section in the main text. You are warned to save all the flow charts you draw as you work exercises in this book because computer problems in the language supplement will refer to them.

Chapter 2

INPUT, OUTPUT AND ASSIGNMENT

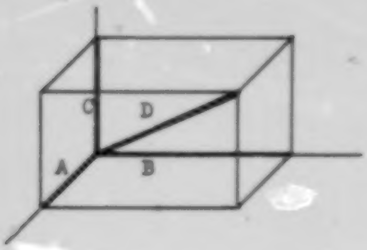
2-1 The Flow Chart Concept

In this chapter we learn about three basic kinds of procedural steps called "input," "assignment," and "output." With these three it will be possible to develop some simple computer programs and in this way gain a progressively better insight for "talking to" or communicating with a computer.

Suppose the instructor has posed the following simple problem:

"Given: $A = 5.0$, $B = 10.0$, and $C = 3.0$.

Find: the 'length', D, according to the formula suggested in Figure 2-1."



$$D = \sqrt{A^2 + B^2 + C^2}$$

Figure 2-1. Diagonal of Rectangular Box

The problem statement might be rephrased in Table 2-1 as a simple, three-step process.

Table 2-1

Procedural phrasing	Corresponding phrasing in original statement
1. Input (i.e., define) the specific values of A, B, and C.	Given A, B, C
2. Using values for A, B, and C defined in Step 1, compute the value for the expression $\sqrt{A^2 + B^2 + C^2}$ and then assign this value to D. A shorthand way of saying all this is: $D \leftarrow \sqrt{A^2 + B^2 + C^2}$.	$D = \sqrt{A^2 + B^2 + C^2}$
3. Output the value of D.	Report the value of D.

This simple sequence of three events, each of which represents an action a computer can perform (provided it receives an appropriate command in a language it can understand), may be expressed even more concisely with a flow chart as shown in Figure 2-2.

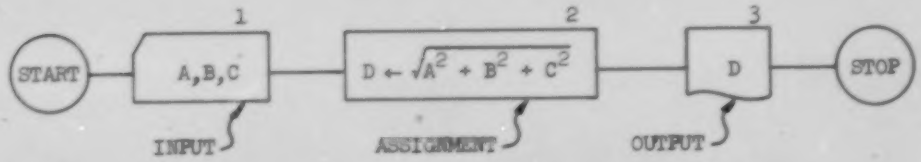


Figure 2-2. Flow Chart for Diagonal of Box

We will usually make it a point to number each box in a flow chart with little numbers placed close to the boxes. It is then easy to refer to any specific part of the flow chart. The numbered boxes in Figure 2-2 represent commands to a computer which could be worded as follows:

1. Input. Read specific values to be assigned to variables A, B, C from a punch card (or other input device) and transfer these values to a pre-assigned location in the computer's memory or storage.

2. Assignment. Obtain the specific values of A, B and C from their storage locations. Compute the corresponding value of $\sqrt{A^2 + B^2 + C^2}$ and assign this value to D. (I.e., put this value in storage at a location associated with the variable D.)

3. Output. Obtain the calculated value of D from its storage position and type or print it on a roll of paper.

[For clarification it should be noted that there are two kinds of input associated with the computing process--input of data, which is what we are discussing; and input of the program (i.e., the set of instructions or commands). Program input will not be discussed in the main body of this text but at any time the student feels the need of knowing how this input is handled or knowing about any aspect of the actual operation of the machine in more detail he may refer to Appendix A.]

The student has probably noticed that nowhere on the flow chart are the values 5.0, 10.0 and 3.0 assigned to A, B and C. This is characteristic (with certain exceptions to be noted) of flow chart writing. We ordinarily do not give on the flow chart the actual numerical values to be assigned to variables but rather indicate how they are to be computed or where they are to be found. One reason for this, as will be seen soon, is that we will use the same section of a flow chart to indicate many repetitions of the same calculation but with different values assigned to the variables.

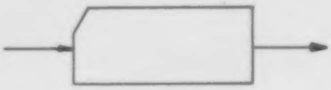
The box labeled 2 in Figure 2-2 is called an assignment step because we assign the computed the value of $\sqrt{A^2 + B^2 + C^2}$ to the variable D. However a great deal more than this is going on in this step. This box contains the indication of the basic computation of the problem which in fact consists of several steps. We will see later how to write more detailed flow charts breaking similar computations into their component parts. Such exercises will help us to see the details or the "fine structure" when we need to. It might seem more reasonable to refer to such a step as the second in Figure 2-2 as a "computation and assignment" step, but we will adhere to the conventional nomenclature of "assignment."

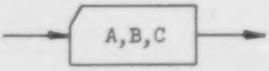
Later in this chapter we shall provide discussion and clarification of the idea of assignment. For now we just observe that our input step also involves "assignment." There we assigned the values on a card to A, B and C. Input always involves assignment but the nomenclature "assignment step" will be reserved for assignment of values which are either computed or obtained from storage.

The *output step* identifies by name the one or more variables whose values, now stored in the computer's memory, are to be *written* out or displayed for us to see.

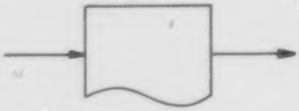
The phrases "*read* a specific value *into* storage" and "*write out* a specific value *from* storage" are often used in speaking about input and output processes. Modern computers are equipped with a variety of input devices which can read data supplied through appropriate input media. Computers at the banks, for example, have input devices which read account numbers printed on checks when these numbers are printed with a special magnetic ink. When typewriters are attached to computers, the data may be supplied simply by typing it.

One of the most popular input devices is the card reader which reads punch cards, so you can see it is natural to adopt as a flow chart convention the silhouette of a punch card to suggest the input process. The list of items being read we shall call the "input list." So to complete the representation of a specific input step, as a flow chart symbol, we insert the input list within the figure, as for example,

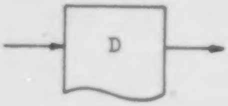




Likewise, some of the most common output devices are line printers and typewriters. These provide us with printed answers in a familiar and readable form. The conventional form representing output is the silhouette



which suggests a piece of paper torn from a typewriter or line printer. Since the key component of the output step is the "output list," to complete the symbol for a specific output step we insert the output list within the figure, as for example,



In this problem, the list is simply D.

There is one more way in which the shape of boxes is indicative of the operations they represent. The circular START and STOP boxes clearly suggest the round buttons commonly used to start and stop pieces of machinery.

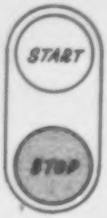


Figure 2-3. Control Switch for Grinding Wheel

From now on we will use the shape of the box to indicate what is going on inside it. This is one of the characteristics of the flow chart language.

Exercises 2-1

In each of the following exercises your job is to convert the problem statement into a flow chart. You will find the structure of each flow chart is similar to that of Figure 2-2.

1. The shaded area in Figure 2-4 is made up of the semi-circle of diameter DE, the square ACDE and the isosceles triangle ABC whose base is b and whose altitude is h. Given b and h, find p, the perimeter of the shaded figure.

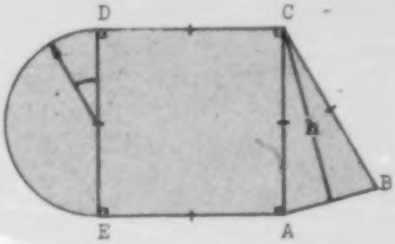


Figure 2-4

2. Given a row M and a column N such that $M \geq N$, find L, the value in the square for the M^{th} row and N^{th} column in Figure 2-5 shown on the right. (Hint: Notice that $8 = \frac{3 \times 4}{2} + 2$.)

	1	2	3	4	5	...
1	1					
2	2	3				
3	4	5	6			
4	7	8	9	10		
5	11	12	13	14	15	
⋮	⋮	⋮	⋮	⋮	⋮	

Figure 2-5

3. Given the grade average, a for n previous homework assignments, and the grade g, for the $n + 1^{st}$ homework assignment, find the average grade, r, based on n + 1 assignments. There are three input values. What are they?

2-2 Repetition

The usefulness of a computer calculation increases if it can be easily repeated. Even the simple, almost trivial, process of Figure 2-2 becomes significant when it is to be repeated a large number of times. Suppose, in fact, the instructor restates the original problem as follows:

"Given: A large number of values of A, B, and C, such as might be found in Table 2-2.

Prepare: A table showing for each set of values A, B, and C (in Columns 1, 2 and 3 respectively) and the corresponding value of D (in Column 4). Assume the same formula for D applies as before."

Table 2-2

A	B	C
5.0	10.0	3.0
4.3	2.5	6.1
8.5	5.7	-3.2
10.4	0.4	0.7
6.3	-5.2	6.4
⋮	⋮	⋮
9.6	17.3	2.2
9.1	-7.2	3.3

While the task involved may comprise far more work than before, the essential change to the process is simply that it be repeated. This concept of simple repetition is very easy to express in a flow chart language by forming a loop, as can be seen in Figure 2-6.

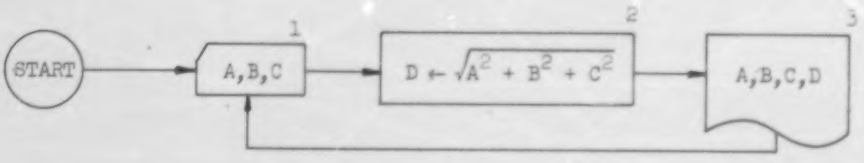
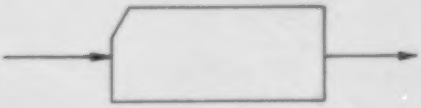


Figure 2-6. A Simple Loop

Such a loop will make sense if we think of each set of data as punched on a separate card, with the cards arranged in a stack, and with only one card read each time the input step is executed. The flow chart tells us something fairly obvious: Instead of halting the process after printing the value of D for the first set of data (along with the values A, B, C), we will return to repeat the entire procedure at the input step (Box 1).

The appearance of the loop in Figure 2-6 requires more exact explanation of the processes represented by our input, assignment and output boxes.



The input box in a flow chart will always have one or more variables written in it. The box can be considered to refer to a stack of punch cards, each with specific numbers appearing on it in the positions belonging to the variables as shown in the flow chart. (See for example Figure 2-7 below.) The command represented by the input box can now be separated into its component parts as follows:

1. Read the numbers off the first card in the stack and put these numbers in storage locations assigned to the variables appearing in the input box. If numbers are already stored in these locations remember that these numbers are completely erased before the new numbers are put in; computer storage devices have destructive read-in.
2. Remove the first card from the stack permitting the next card (if any) to become the first card.
3. If a flow chart arrow carries us into an input box and it turns out that there are no cards left in the stack then the computation is to stop.

Without the last part of the above explanation Figure 2-6 would suggest an "endless loop." Such a loop would represent a most unsatisfactory algorithm. We now see that the computation in Figure 2-6 is provided with a way to stop; there will not be an infinite loop unless there are infinitely many cards.

Inside an assignment box we will always put a left-pointing arrow. To the left of the arrow we always put a variable. To the right of the arrow will appear some sort of expression; it may be a constant; it may be a variable or it may be, as in the previous example, an expression indicating a computation. The breakdown of the command of the assignment box follows.

1. For each of the variables (if any) occurring on the right side of the arrow in the assignment box, read-out the value from the appropriate location in storage. This read-out is non-destructive. That is, the values to be found in the locations corresponding to these variables are exactly the same after this read-out as before. As described in Section 1-3, read-in is destructive, while read-out is not.
2. Once the values of the variables appearing on the right side of the assignment box have been read, any computation indicated by the expression on the right hand side of the arrow is performed.
3. The value of the expression on the right side is assigned to the variable on the left side of the arrow. That is, the result of the computation is read-in (destructively) to the storage location corresponding to the variable on the left side of the arrow.

As an example suppose that the input of the process illustrated by Figure 2-6 consists of the three cards in Figure 2-7 (in the indicated order). Then the output will be as in this figure.

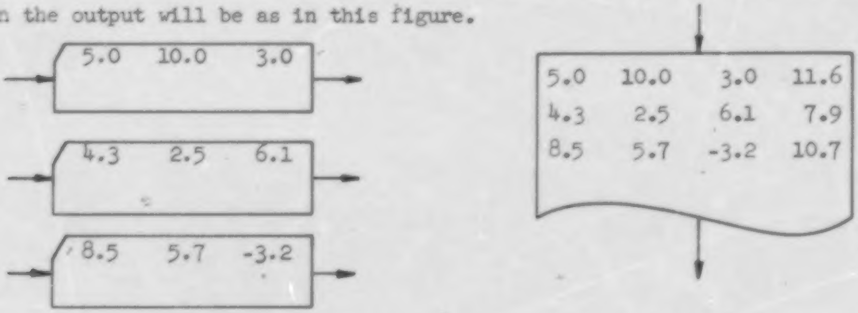


Figure 2-7. Sample Input and Output Data

The student may wonder why the values of A, B, C should be printed as output when they were known at the beginning. Suppose only the last column in Figure 2-7 were printed as output. The person reading this output would then know for example that 7.9 is $\sqrt{A^2 + B^2 + C^2}$ for some values of A, B and C which he might be able to find if the order of the cards has not been disturbed, and provided the cards can be located. The value of this output data would obviously be seriously reduced.

Exercise 2-2

For the flow chart of Figure 2-6, repeated in Figure 2-8 below, with the input of Figure 2-7, give the values stored in the positions corresponding to each of the variables A, B, C and D at the indicated point in the computation.

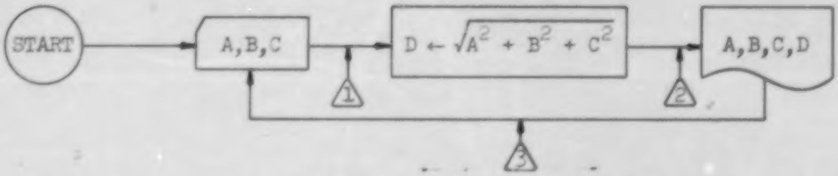


Figure 2-8. A Simple Loop

(a) At position △1 when no output has been printed.

(b) At position △2 when two lines of output have been printed.

(c) At position △3 when one line of output has been printed.

(d) At position △1 when one line of output has been printed.

2-3 Assignment and Variables

The concept of assignment is of fundamental importance in computing, so we should really subject it to the most careful scrutiny. Assignment is quite different from any concept you have met in mathematics, although it is similar enough to be mistaken for either "equality" or "substitution." We will explain how assignment differs from these concepts at the appropriate time.

As has been stated in the previous section, we always assign values to variables.

A variable in mathematics is usually a letter which has not been previously identified as a constant. A letter followed by one or several integer subscripts is also permissible. In computer language we allow a great deal more leeway in the symbols which may be used as variables. Some samples are:

A, B, C, X, Y

and such descriptive combinations of letters as:

DIST, AREA, LENGTH, ARGGH,

or such strings of letters and digits as:

A_3, X_2, Y_3G_5, R_3C_4 .

There are two principal reasons for enlarging our list of variables to include these strings. First, the list of symbols available to computers is usually limited to upper case Roman letters. There are no Greek letters and usually no lower case letters. We just do not have enough letters available for use as variables. Second, using a descriptive combination of letters as a variable is often very helpful in reminding us how the variable is being used.

We have a special attitude toward such unbroken strings of letters and digits starting with a letter. We regard them as being connected together to form a brand new symbol, somewhat like handwriting. We think of symbols above as being written as follows:

DIST dist
R_3C_4 R3C4

The occurrence of punctuation, operation symbols, or parentheses breaks the spell. The above attitude applies only to strings of letters and/or digits commencing with a letter. Any such string of characters will be regarded as a variable unless there has been a specific statement to the contrary. From this point of view an expression like XN is *not* considered to contain either of the variables X or N but rather to be a brand new symbol. In other words we insist no variable should be considered to be part of another variable.

Now we are able in a few words to explain the use of variables in computing and the idea of assignment.

In any computing problem, there corresponds to each variable used in that problem a location in the computer's storage. By *assigning* a number to a variable we mean simply reading the number destructively into the storage location corresponding to that variable. In evaluating arithmetic expressions a variable is to be treated as a name for the number in the corresponding storage location. The number in the corresponding storage location is referred to as the *value* (or *current value*) of the variable. During the course of a computation many different values (perhaps even millions) may be assigned to a given variable. Thus it will not be meaningful to speak of the value of a variable without specifying the time or, more precisely, the stage of the computing process. But once the stage of the process is specified, the value of the variable is uniquely determined. (See, for example, the exercise at the end of the preceding section.)

A storage location may be hard to visualize. If so, here is an analogy which will not lead to error. Consider that to each variable there corresponds a wooden box. To make the correspondence clear we engrave on the boxes the corresponding variables. (But remember that the variable is a name not for the box but for the number inside.)

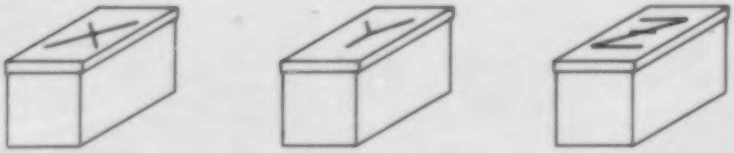
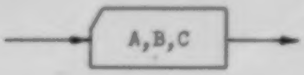


Figure 2-9. Three Boxes with Identification

To assign 2.5 to the variable X, we open the box labeled X, dump out the contents and put in 2.5. We will speak for a while in terms of these boxes.

Assignment may be done either in an input step or in an assignment step. When, as in the example of the preceding section, we come to the input box,



we empty out the boxes labeled A, B and C and fill them respectively with the values punched on the proper input card.

You will remember that an assignment box has the form shown in the following figure.

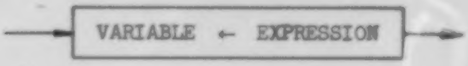


Figure 2-10. Form of an Assignment Box

A left-pointing arrow is used to avoid confusion with the many uses of right-pointing arrows in mathematics.

Immediately we see some inadmissible forms for assignment boxes as in Figure 2-11.

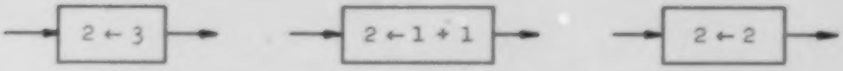


Figure 2-11. Inadmissible Assignment Boxes

The reason that these assignment statements are inadmissible is that a constant rather than a variable appears to the left of the arrow.

Another inadmissible assignment box is shown in Figure 2-12.

$$\longrightarrow \boxed{B^2 - 4 \times A \times C \leftarrow 5} \longrightarrow$$

Figure 2-12. Another Inadmissible Assignment Box

Again the expression on the left is not a variable and we cannot assign anything to it.

Now we are ready to examine some admissible assignment boxes. The simplest form,

$$\longrightarrow \boxed{X \leftarrow 2} \longrightarrow$$

is interpreted: Dump out the value in the box labeled X and put in 2.

We hasten to remind the student that assignment is not equality. Some beginners erroneously read $\boxed{X \leftarrow 2}$ as "X = 2." Then, later in the process they may see $\boxed{X \leftarrow 3}$ and think, "X = 3." Combining these two statements they have $2 = 3$ which points out the confusion. Of course if we see $\boxed{X \leftarrow 2}$ and later $\boxed{X \leftarrow 3}$ we remember that the number 2 is cleared out of the location associated with X before 3 is put in. Thus we do not imply that $3 = 2$ but merely that 2 and 3 were consecutive tenants of the location belonging to X.

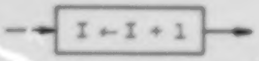
Now consider the assignment statement:

$$\longrightarrow \boxed{X \leftarrow T} \longrightarrow$$

This statement does not mean: Go through all the formulas involving X and replace X by T. This is another way in which beginners go wrong. Such an interpretation does not yield correct results.

What the above assignment statement does mean is the following: Go to the box labeled T and read the value contained therein (but do not alter this value). Empty out the box labeled X and put into it the value read out of the box labeled T. As an example, suppose the values of X and T were 5 and 7 respectively before the command X ← T is carried out. After this command is executed the values of X and T will both be 7.

An assignment statement we frequently see is,



To execute this command we go to the box labeled I and read the value contained therein. Then we add 1 to this value. Now we empty out the contents of the box labeled I and put in the value just computed. As an example, if the value of I was 7 before the execution of this command, the value of I will be 8 after the command is executed.

This last assignment box can be thought of as an "updating" rule (if I represents the date). Assignment steps of this kind are often used when the incremented variable is being used as a tally or counter.

As an example of this idea, suppose that in the flow chart of Figure 2-6 with the input of Figure 2-7 we wish the lines in the output to be numbered in order. Figure 2-13 is a revised flow chart achieving this result.

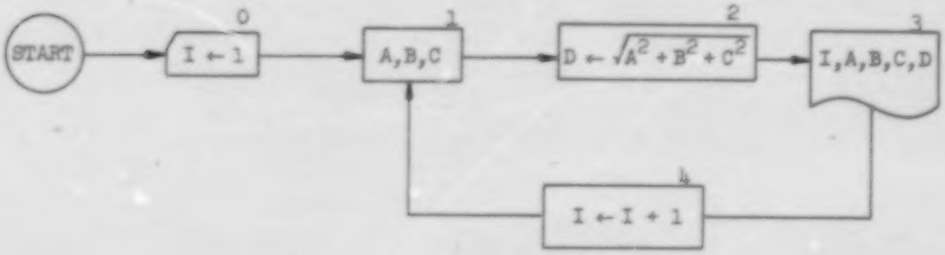


Figure 2-13. Revised Flow Chart for Problem of Preceding Section

The "counter," I, keeps track of the number of sets of data, A, B, C, read in to memory and used to compute D. We begin by assigning to I an initial value of 1 (Box 0). Each time we print out the results, we increment or "update" the value of I (Box 4). Notice that the value of I will be printed along with the values assigned to A, B, C and D each time the output step (Box 3) is executed. If the input of Figure 2-7 is fed into the process of Figure 2-13 then the output will be as in Figure 2-14.

```
1    5.0    10.0    3.0   11.6
2    4.3     2.5    6.1    7.9
3    8.5     5.7   -3.2   10.7
```

Figure 2-14. Sample Output Data from Preceding Flow Chart

In Chapter 3 you will see how the same updating step $\rightarrow\boxed{I \leftarrow I + 1}\rightarrow$ may be used to control a repetitive process like that in Figure 2-6, but here we are using this step only to keep track of or to "monitor" the repetitive process.

Let us now make an improvement in the box we visualize as corresponding to a variable. From now on think of the box as having a window in it. All read-out will be done through this window. This will eliminate the danger of altering or destroying the number in the box by our reading process. To avoid confusion with input, output and assignment boxes we will from now on refer to these wooden storage boxes as "window boxes."

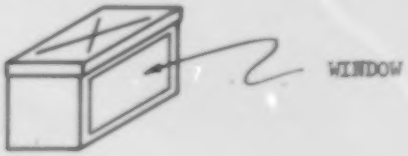


Figure 2-15. The "window box"

The window box with the letter X on it will be opened only

1. during an input step when X appears in the input box, or
2. during an assignment step when X appears on the left side of the assignment arrow.

Thus a window box is opened only when reassignment is to take place. Once a value is assigned to a variable, its corresponding window box will henceforth never be empty.

The following two assignment commands are easily interpreted.

$$\longrightarrow \boxed{T \leftarrow \sqrt{T}} \longrightarrow$$

Read through the window the value in the box labeled T. Compute the square root of this value. Empty out the value in the box labeled T and replace with the computed value.

$$\longrightarrow \boxed{\text{DISCRIMINANT} \leftarrow B^2 - 4 \times A \times C} \longrightarrow$$

Here we read the value of A, B, and C. Using those values we compute $B^2 - 4 \times A \times C$. Now we read this computed value (destructively) into the box labeled DISCRIMINANT.

You may find it helpful to think of the work of a computer as being done by three people, the "master computer" and two assistants called the "assigner" and the "reader." When the master computer wishes to assign a value to a variable, he writes the value on a slip of paper. He gives this slip of paper to the assigner and tells him which variable to assign it to. The assigner finds the appropriate window box, empties out the contents and puts in the slip of paper bearing the new value.

When the master computer wishes to know the value of a variable he calls the reader and tells him which variable to look up. The reader goes to the appropriate window box and, looking through the window, makes a copy of the value inside the box on a small pad of paper. He then returns to the master computer and gives him the sheet of paper bearing the copied value.

The "master computer" has a master, too. It is you who write the algorithms which he must run. An algorithm will be poorly formed if, in carrying it out, the master must send a "reader" to a window box before sending an "assigner" to that same box. Why?

Exercises 2-3

1. The following is a restatement of Exercise 2-1, Number 3:

   Given the grade average, OLDAVG, for n previous homework assignments and the grade, GRADE, for the $n + 1^{st}$ homework assignment, find the new average, NEWAVG, based on $n + 1$ assignments.

   Convert this problem statement into a flow chart with a structure similar to that of Figure 2-6.

2. Compare the above problem statement and your solution of it with the original statement and your solution to that one. Which flow chart would convey more meaning to you after one week, one month, one year, if you were to glance at each one without looking at the corresponding problem statement? Comment on the lesson that is to be learned here.

3. Suppose we modify the problem in Exercise 1 as follows.

   The new average grade for $n + 1$ homework assignments is to be assigned not to NEWAVG, but to OLDAVG. Redraw the flow chart for this case. Why is this a reasonable thing to do?

4. The instructor began keeping two pages in his gradebook for each class. On Page 1 he recorded the actual scores or grades for each of the homework assignments. This page isn't shown here. On Page 2 he recorded the new cumulative grade averages, column after column, as they were computed following the grading of each assignment. Page 2 is illustrated in partial detail below.

Cumulative Grade Averages

Name	CUM 1	CUM 2	CUM 3	...	CUM 7	CUM 8
Abel, John	79.0	81.5	83.0	...	77.1	
Baker, Tim	83.0	84.5	84.0	...	83.4	
Chary, Smiley	54.0	64.0	67.0	...	71.2	
⋮						
Thompson, Bill	83.0	81.5	84.0	...	83.1	
Williams, Ted	81.0	83.0	80.0	...	84.6	

Note that each column shows the average based on one more score than the previous column. Thus, after 3 scores, Tim Baker had an average of 84.0. After 7 scores, his average was 83.4.

To compute each column we imagine the instructor follows a manual or computer process based on the flow chart you just developed in Exercise 3. Now suppose, the grades for the eighth homework set are

John Abel	91
Tim Baker	88
Smiley Chary	82
⋮	⋮
Bill Thompson	88
Ted Williams	87

What are the input data values to compute column CUM 8 entries for John Abel, Smiley Chary and Ted Williams?

5. After some experience with this grade recording system the instructor felt it no longer necessary to maintain Page 1 information showing the individual grades. Having the series of cumulative averages seemed adequate for him. (On only rare occasions did he need to look at an individual grade). In your opinion was the instructor safe in dropping Page 1 from his records? Explain.

6. Develop a flow chart showing the process by which any desired grade can be computed from Page 2 information alone.

2-4 Arithmetic Expressions

We have to take a close look at our mathematical notation and we commence the scrutiny in this section, focusing our attention on those rather vague "expressions" on the right hand side of assignment boxes. These expressions usually represent an indicated calculation and we will refer to them as "arithmetic expressions." It should be noted that the term "arithmetic expression" has a larger scope in computer work than in mathematics.

Although our usual (every day) mathematical notation is very useful and flexible and quite adequate for our mathematical needs, it is not suitable for mechanical reading. This is what we mean when we say that ordinary mathematical language is not a "formal" language. Applied to arithmetic expressions this means that we cannot write down sets of rules for:

1. determining whether or not any arrangement of symbols constitutes an arithmetic expression;
2. telling how to evaluate any arithmetic expression we may be given.

And yet, with a few small changes in our mathematical notation, the language of arithmetic expressions is converted into a language capable of being formalized. We indicate what these changes are and give the formal rules for evaluating expressions, after these changes have been made. Part of the rules for determining whether an arrangement of symbols is an arithmetic expression (that part concerning detailed study of the use of parentheses) is left to Appendix B.

This discussion of the modifications necessary to formalize mathematical language should be of considerable help to you in your programming work. Most of the programming languages you are likely to be studying are based on everyday mathematical language. The material of this section should provide a "rationale" for the departures from everyday mathematical usage encountered in those programming languages. Some of the "reforms" proposed in this section we do adopt in the flow chart language. Others we do not. The ones not adopted do, however, provide us with an alternative way of writing things if problems in readability occur.

A number of years ago at a large American university, an entrance exam contained the question:

Simplify

$$\frac{\mathrm{sinx}}{\mathrm{tanx}} .$$

One student gave as his answer:

$$\frac{si}{ta} .$$

The professors involved in the grading of the test had a good chuckle over this unexpected misinterpretation of the problem.

Unexpected though it may have been, still there was nothing ridiculous in this answer. Presumably the student had not studied trigonometry and according to the rules learned in his algebra course his work was perfectly correct. He had no other way of interpreting sinx but as

$$s \times i \times n \times x .$$

If the problem had been

Simplify

$$\frac{abnx}{cdnx}$$

there could be no alternative to the answer

$$\frac{ab}{cd}$$

The student here was using a rule which he believed to be permanent and unchanging; but, alas, it had been superseded by another rule. In fact, it may not have been entirely superseded. Suppose we were dealing with a problem not involving trigonometry in which

$$s, i, n, \text{ and } x$$

were variables. Might we not then regard

$$sinx$$

as denoting

$$s \times i \times n \times x \text{ ?}$$

Next, consider the frustrating expression

$$|a|b|c| .$$

Which does it denote,

$$|a| \times b \times |c| \quad \text{or} \quad |a \times |b| \times c| \text{ ?}$$

According to our usual convention of using juxtaposition to denote multiplication, either of these last two expressions may be written in the form $|a|b|c|$. Yet if b is negative, these expressions will in general have different values.

These examples show that the meaning of mathematical expressions is sometimes ambiguous and may depend on context for correct interpretation. In the first example the use of juxtaposition for multiplication is the culprit. In the second, juxtaposition conspires with the indistinguishability of left and right absolute value symbols to rob the expression of its meaning.

We hope that we have not given the impression that mathematical notation is unsalvageable. In truth, practices current in everyday mathematical notation which may lead to ambiguity, or difficulties of formalization, are few in number. We list the "reforms" needed or helpful to insure absolute clarity in the mathematical language of expressions. These reforms will be used throughout this section, thus obtaining a language rather like a programming language. When we return to ordinary flow chart language in Section 2-5, only the first and fifth reforms noted below will be retained.

✓ 1. Abandon the practice of using juxtaposition to denote multiplication and instead use the operator symbol "×." We will do this in flow chart language.

2. Special functional notations cause trouble because, in giving formal rules for reading, it is necessary to give a special rule for each such notation. To avoid this, replace these notations by such notations as ABS(X) for $|x|$ and SQRT(X) for $\sqrt{x}$. In flow chart language, we will continue to write $|x|$ and $\sqrt{x}$ since the human reader generally finds them easier.

3. Abandon off-the-line notations because they strike a death blow at any hope of a simple formalization procedure if the part occurring off the line is allowed to have variables in it. Substitutions can then carry farther and farther off the line, giving rise to numerous types of difficulties. We adopt here the notation

$$x\uparrow 3 \text{ for } x^3 .$$

In flow chart language we will stick to superscripts.

4. Abandon use of "-" in three different senses: as a binary subtraction operator in "X - Y"; as a unary "taking the negative of" operator in "-X"; as part of the name of a negative constant in "-5." †

✓ 5. Embrace function arguments in parentheses. We will also do this in the flow chart language except in the special notation listed under (2).

6. Adhere to the usual conventions regarding parenthesis removal. This is discussed later in this section and also in Appendix B.

Now all the valid arithmetic expressions in our "modified" language can be generated by using Table 2-3 together with the rule that follows.

Table 2-3

Basic Forms of Arithmetic Expressions

Kind	Examples
1. Numerical Constants	17, .0065, 3.14159, 0 -5, --.061, -17.62 *
2. Variables	X, Y, A, B, DIST, AREA, ARGGH
3. Unary Operational Form	-X *
4. Binary Operational Form	X + Y, X - Y, X × Y ** X/Y, X↑Y
5. Functional Forms	SIN(X), COS(X) ABS(X), SQRT(X)

[The asterisks occurring in the table will be explained as we go along.]

---

† The three uses of "-" were introduced into mathematical notation in order to profit from the confusion. Although they cause quite a lot of trouble, we retain the three minuses. An example showing the three minuses is

$$\text{GLDLKS}\ \underset{\text{binary}}{-}\ (\underset{\text{unary}}{-}(\underset{\text{number-naming}}{-}5))$$

Does "profit from confusion" confuse you? It shouldn't. How were we taught to recognize at a glance that A - (-(-5)) is the same as A - 5? Answer--When we want to simplify an expression we treat all three different minuses as if they were the same--in the rule that an odd number of them can be replaced by a single one.

In our computing work we will take what may seem a narrow view of what we consider to be numerical constants. By numerical constants we mean strings of digits with or without a decimal point and possibly preceded by a minus sign. That is all; no exceptions. For example, we do not consider

$$2 + 5 \quad \text{and} \quad 3/4$$

as numerical constants, but as expressions still to be evaluated: indicated operations to be carried out. Some writers do not permit the functional forms (entry 5 in Table 2-3) to be classified as Arithmetic Expressions.

Along with the table goes a rule for grinding out more and more expressions.

> RULE: In an arithmetic expression, if a variable is replaced by an arithmetic expression, the result is again an arithmetic expression.

A slight modification of this rule is necessary in light of the following example: Suppose in $B \times X$ we replace X by $-A$. We then obtain $B \times -A$. This juxtaposition of two operator symbols ("×" and "-") is not permitted in mathematical writing and it is not permitted in most computer languages either. We must in this case put parentheses around $-A$ then obtaining $B \times (-A)$.

One might wish to be able to apply the above rule directly without making any exceptions concerning parentheses. This result could be obtained by going back to Table 2-3 and putting parentheses around the basic forms in the boxes marked with a single asterisk (*). In this way we would always have parentheses written around negative constants and unary operational forms, such as

$$(-5),\ (-.0091),\ (-X),\ (-A).$$

We are used to having the replacement or substitution in the above rule do more for us than merely produce a valid expression. We will illustrate with these examples:

Example 1:

If we substitute 3×5 for X in the expression $2 + X$ we obtain $2 + 3 \times 5$ which we evaluate according to our usual rules to be 17. If, on the other hand, we evaluate 3×5 and substitute the result for X in $2 + X$ we obtain $2 + 15$ which we evaluate to be 17. The two substitutions made for X produced equivalent (i.e., "equal-valued") expressions,

$$2 + 3 \times 5 \text{ and } 2 + 15.$$

But now let us look at another example.

Example 2:

If we substitute $3 + 5$ for X in the expression

$$2 \times X$$

we obtain

$$2 \times 3 + 5$$

which evaluates to 11. If, on the other hand, we evaluate $3 + 5$ as 8 and substitute this value for X in

$$2 \times X$$

we obtain

$$2 \times 8$$

which evaluates to 16. The two substitutions made for X produced the expressions

$$2 \times 3 + 5 \text{ and } 2 \times 8$$

which are not equivalent.

No doubt every reader has spotted the trouble; we left out the parentheses. In every mathematical use of replacement or substitution in the above rule we want the results of the two orders of doing things to be equivalent. This is what is indicated by the old maxim, "When equals are substituted for equals the results are equal." In order to attain this end we put parentheses around the $3 + 5$ in the last example to obtain

$$2 \times (3 + 5)$$

thus ensuring the desired order of computation.

We might wish to preserve this property of obtaining equivalent expressions through use of our rule without making special cases concerning use of parentheses. If we so wish we can attain the desired result by again modifying Table 2-3 by putting parentheses around the forms in the box labeled with double asterisks (**).

Suppose we were to put parentheses around the forms in the boxes labeled with asterisks. Then as we used our rule to generate more complicated expressions we would find many more parentheses occurring than we are accustomed to write. For example, we might find

$$((X + (Y \times SIN((A +(B \times C))))) + (5 \times(X + A)))$$

where we would ordinarily write

$$X + Y \times SIN(A + B \times C) + 5 \times (X + A)$$

These two expressions are equivalent because of our agreement on the order in which operations are to be performed in the absence of parentheses.

If we were to put all these parentheses in Table 2-3 we should then have to give a rule for removal of parentheses in conformity with ordinary usage. This rule is rather complicated and we do not feel it necessary to discuss it here. (It can be found, however, in Appendix B, for those who are interested.) There is a wide agreement (exhibited in the precedence table below) on the order in which operations are to be performed in the absence of parentheses. We will assume that you have mastered the art of writing expressions, putting in parentheses where necessary to indicate the order of calculation. The scanning process used by computers is designed to use the same order of calculation. In other words, it performs calculations in the same order you would. We will exhibit this order of computation. After that it will be up to you to write in such a way that the computer will carry out your intent. One last word of admonition: If in doubt whether parentheses are necessary, *put them in!*

You have long known that in expressions such as

$$2 + 3 \times 4$$

the multiplication is to be performed first. We convey this information by saying that multiplication is more cohesive or more binding than addition or perhaps we say that multiplication takes priority or precedence over addition. This kind of information is collected in Table 2-4. We have taken the liberty of using the symbol "↑" for exponentiation. In the flow chart language we will still use superscripts.

Table 2-4

Precedence Levels for Evaluating Parenthesis-Free Expressions

Level		Operation Name	Operator Symbol
High	First	Exponentiation	↑
↕	Second	Multiplication	×
		Division	/
		Taking the Negative	- (unary)
	Third	Addition	+
Low		Subtraction	- (binary)

In evaluating parenthesis-free expressions you (or the computer) first scan from left to right for operators of the first level. If none are found, scan (left to right) for operators of the second level. If no operators of the second level are found, then scan for operators of the third level. As soon as you locate an operator of the type being scanned for, perform the operation that it indicates. Then remember where you were in the scanning process and take up from there. (A simpler rule to state and one which could equally well have been chosen is to restart every scan at the beginning of the expression.)

We give an example showing how this all works out.

Example

The expression is

$$I - N \times A \uparrow N / D + Ø \times U - T$$

Tabulated values for the variables:

A	D	I	N	Ø	T	U
2	4	9	3	7	9	4

Table 2-5 displays the step by step evaluation. Little triangular symbols (▲) are used to indicate the operator symbol to be dealt with next.

Table 2-5

Display of Step by Step Evaluation

Example 1

Step No.	Action	Appearance of the Expression After Each Step	Remarks
	Initial appearance	I - N × A↑N/D + Ø × U - T ▲	
1	Compute A↑N	I - N × 8 /D + Ø × U - T ▲	No more level 1
2	Compute N × 8	I - 24 /D + Ø × U - T ▲	
3	Compute 24/D	I - 6 + Ø × U - T ▲	
4	Compute Ø × U	I - 6 + 28 - T ▲	No more level 2
5	Compute I - 6	3 + 28 - T ▲	
6	Compute 3 + 28	31 - T ▲	
7	Compute 31 - T	22	

You may wonder how to tell a unary minus from a binary minus. Do they have little tags on them? No, but in a properly written arithmetic expression a unary minus can occur either at the very beginning of an expression as in -Y, or immediately following a left parenthesis and nowhere else. A binary minus, on the other hand, can never occupy such positions.

The scanning process shown in Table 2-5 constitutes the heart of the evaluation process. We now finish the description of evaluation by explaining what to do with expressions containing parentheses. A "sub-expression" of an expression is defined to be any part of the expression included between a pair of parentheses. For example, in the expression

(A × C - D) × E

we see that

A × C - D

is a sub-expression.

Table 2-6 gives the procedure for evaluating an expression with parentheses.

Table 2-6

Rules for Evaluating Arithmetic Expressions with Parentheses

1. Scan expression from left to right for first right parenthesis ")".
2. Evaluate the sub-expression ending with this right parenthesis according to the rule for parenthesis-free expressions. (Table 2-4).
3. If this sub-expression is a constant, see whether it is preceded by a function name, and if so, compute the indicated functional value.

[NOTE: Parentheses surrounding a constant should be deleted if possible. In the scanning procedure the (undeletable) parentheses, surrounding a negative constant but not preceded by a function name, should be ignored. In such a case this negative constant together with its surrounding parentheses is to be treated as a numerical constant.]

We must confess that there is one place where our instruction for the order in which expressions are to be read gives results not in conformity with usual mathematical conventions. This is in expressions of the form

$$A^{B^{C}} \quad \text{or} \quad A \uparrow B \uparrow C \; .$$

What is the value, for example, of

$$2^{3^{3}} \; ?$$

It can be either 512 or 134,217,728 depending on how parentheses are inserted. As you can imagine, in certain calculations the difference between these two values may be of considerable importance.

The rule given in the text would evaluate

$$A \uparrow B \uparrow C$$

in the order

$$(A \uparrow B) \uparrow C \; .$$

In mathematics, however, the convention is that

$$A \uparrow B \uparrow C \quad \text{means} \quad A \uparrow (B \uparrow C).$$

Or in customary mathematical notation

$$A^{B^C} \quad \text{means} \quad A^{(B^C)} .$$

Be sure you are aware of this discrepancy. You can always force your intent by use of parentheses.

Exercises 2-4 Set A

1. Create a table of "step number" and "action" similar to the first two columns of Table 2-5 for the step by step evaluation of the expression

$$((a \times X + b) \times X + c) \times X + d$$

where values of the variables are

a	b	c	d	X
2	-1	2	-3	2

2. For the expression

$$(a - b) \times (c - d)/(e \times (f + g))$$

where values of the variables are

a	b	c	d	e	f	g
1	2	3	4	3	2	1

Create a "step number" and "action" table, as in Problem 1.

3. For the expression

$$\frac{3.14}{2} \times r^2 - (s \times \sqrt{r^2 - s^2} + r^2 \times \text{PHI})$$

where $r = 10$, $s = 9$, and $\text{PHI} = 1.12$, write down the "action" of the 4th, 8th and 12th steps in the step by step evaluation.

4. For the expression

$$\sqrt{\frac{1}{2}(1 + q/\sqrt{p^2 + q^2})}$$

where $p = 3$, and $q = 4$, write down the "action" of Step 5 in the step by step evaluation.

5. If the expression in Exercise 3 were modified to read

$$\frac{3.14}{2} \times rsq - (s \times \sqrt{rsq - s^2} + rsq \times PHI)$$

how many fewer steps would be required for its evaluation? Here rsq presumably is a variable whose value is the square of r assigned in a step prior to the evaluation of the given expression.

We have suggested no limit to the complexity of the mathematical expressions which appear in assignment steps. Practically speaking, there is only the limit of the eye's ability to scan and the mind's ability to analyze for unique meaning.

Some expressions like

$$A + B$$

or

$$A + B \times C$$

or

$$A + B \times C/(2.0 + F)$$

involve only horizontal inspection left-to-right. Since only one direction is involved, we speak of this as a one-dimensional or "linear" scan.

These "horizontal" expressions are really just strings of characters. If we can get these character strings into memory in consecutively addressed positions then the computer can be programmed to inspect and interpret them as expressions. The rules of precedence within subexpressions which we studied in Table 2-4 then govern the computer's interpretation procedure.

What kinds of expressions can we transmit to memory as character strings?

In a sense, the statements of our procedure, when thought of as character strings, are just another form of input data, so the manner of transmission depends on the input media available. For the sake of simplicity we continue to assume the punch card is our input medium. However, most of the ideas discussed below are applicable to other input media, like punched paper tape, typewriter keyboards, etc.

Figure 2-16 shows the expression

$$A + B \times C/(2.0 + F)$$

as it might appear on a punched card.

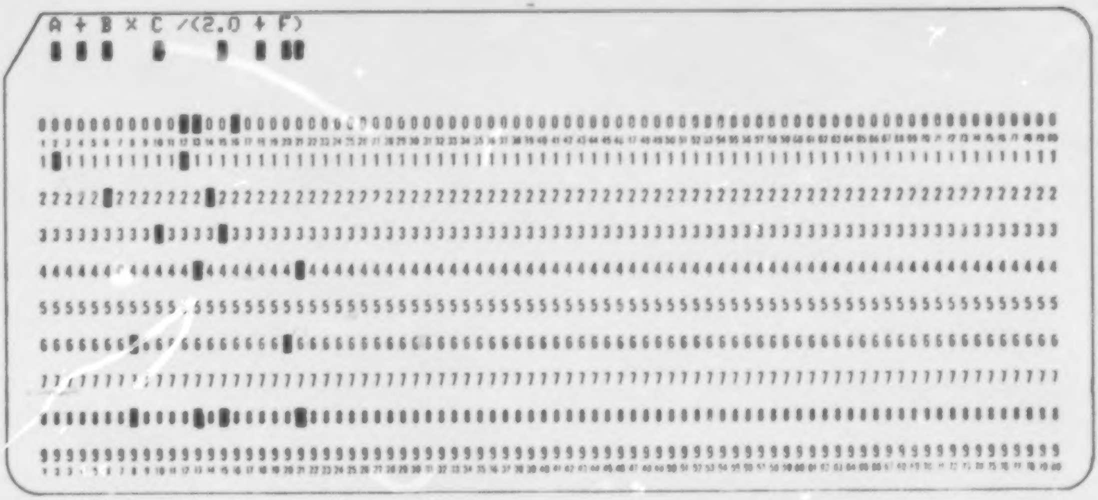
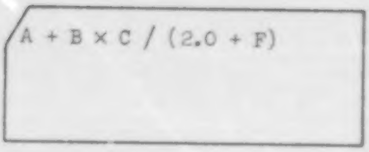


Figure 2-16. Punch card with an arithmetic expression[†]

Each column of the card contains punched information or is blank; i.e., no holes. The input device, when properly activated, automatically transfers the information on the card to the computer's memory as a string of characters. Figure 2-17 indicates two ways this information might be stored in consecutively addressed positions.

[†] The character set shown here is not identical with that of Figure 1-18.



101 A	102 +	103 B	104 ×
105 C	106 /	107 (	108 2
109 .	110 0	111 +	112 F
113)	114	115	116

a. Characters stored one per word of memory

55 A + B	56 × C /
57 (2.	58 + F)
59	60

b. Characters stored three per word of memory

Figure 2-17. An expression in memory

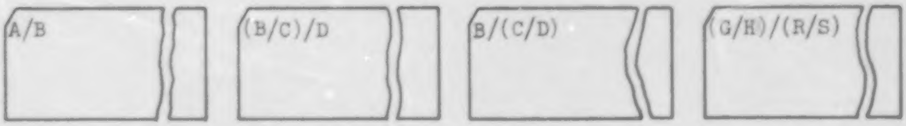
From now on, when we speak of a computer scanning an expression, we shall mean that the expression, originally punched on the card or transmitted via some other input medium, is examined one character at a time from the section of memory where it is stored.

Thus, a left-to-right scan corresponds to a sequential examination from lowest to highest memory addresses. In many machines, characters can be grouped two, three, or more to each word or address. In such cases a left-to-right scan amounts to looking at the characters of one word from left-to-right and then examining the characters at the next higher address in the same way, continuing word after word until the last character has been examined.

Other expressions like

a. $\frac{A}{B}$, b. $\frac{\frac{B}{C}}{D}$, c. $\frac{B}{\frac{C}{D}}$, d. $\frac{\frac{G}{H}}{\frac{R}{S}}$

also involve one dimensional inspection. We are accustomed to rely on our eye for a vertical scan, but there is no direct analogy with the punch card unless we adopt conventions to convert vertically-written expressions to equivalent horizontal forms which preserve unique meaning. Some possibilities are shown in Figure 2-18.



Case a   Case b   Case c   Case d

Figure 2-18. Examples of expressions written horizontally

It is necessary to replace bars (———), representing division, with slashes (/) when we transform a vertically-arranged expression to horizontal form for the punch card.

Parentheses may be introduced to preserve the meaning usually clearly understood in the vertical display. Are the parentheses used in Cases (b) and (c) of Figure 2-18 really necessary? If you are in doubt of the answer, review the rules of Table 2-6.

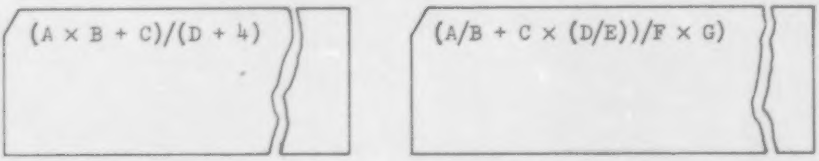
As expressions become more complex the eye is expected to travel first in one direction, then in another. Expressions like

$$\frac{A \times B + C}{D + 4} \quad \text{or} \quad \frac{\frac{A}{B} + C \times \frac{D}{E}}{F \times G}$$

are good examples. Figure 2-19 suggests how these may be converted to horizontal form. Again, parentheses are employed to reduce the risk of ambiguity.

Exercise 2-4 Set B

Not all the parentheses used in Figure 2-19 are necessary. Applying the principles developed in this section, can you identify the parentheses which are superfluous?



Case a   Case b

Figure 2-19. Expressions which may have unnecessary parentheses

2-5 Rounding Functions

There is a class of mathematical functions known as integer rounding functions which are of special interest in computing. Our interest in rounding functions comes from two sources:

1. Every arithmetic operation on real numbers in a computer implies the use of some rounding function. To understand the effect of arithmetic operations we should, therefore, be familiar with these functions.
2. Rounding, properly interpreted, is often a key step in the solution of problems, and therefore, in the design of algorithms. What is meant, for example, by an instruction for half the class to go to the blackboard? If there are 25 in the class, should 12, $12\frac{1}{2}$ or 13 go to the blackboard? Remember, in developing algorithms we must be unambiguous.

An integer rounding function has a real number for its argument and it yields an integer value for its result. The integer obtained is, in some sense, a "best" approximation to the given real number. Each function in this class embodies a different interpretation of the word best. For example, one integer rounding function (called the "greatest integer function") yields the value 1 as the best integer approximation to 1.6.

We know from our study of Section 1-4 that computers often have the capability to perform arithmetic operations efficiently on real numbers over a very wide range when coded in floating-point form (i.e., exponent and precision parts). The results while not perfect are correct to the last place. Furthermore, computers perform integer arithmetic operations yielding exact results provided the integer operands don't get too big. It is therefore often of great advantage to convert numbers, stored in floating-point form, to some integer representation, and vice versa. When studying programming languages, such as FORTRAN, or ALGOL, we will be learning ways to tell the computer how to convert numbers in memory from one representation to another. Because our flow chart analogies must be capable of describing any computer action, including that of rounding, we need a mathematical notation to express precisely this action inside the boxes of a flow chart.

The Greatest Integer Function

A particular integer rounding function which is very simple, of frequent occurrence in mathematics, and of fundamental importance in computing, is the "greatest integer function." The usual mathematical notation for this function is

$$[x]$$

which is subject to many, but not all of the criticisms of the absolute value notation. One possible alternative notation is

$$\mathrm{GRIN}(x)$$

from GReatest INteger.[†] The usual mathematical notation will be adequate for flow chart language.

The function is defined as follows:

$[x]$ = the greatest integer which does not exceed x .

The value of this function can be explained geometrically by considering x to be a point on the number line. To find $[x]$ we start out at x and,

1. if x is an integer we stay where we are, while
2. if x is not an integer we move to the first integer to the left.

In this way our final location is at $[x]$. We see in Figure 2-20 some examples of this procedure. Thus, the figure tells us that $[-2.1] = -3$, while $[2.1] = 2$.

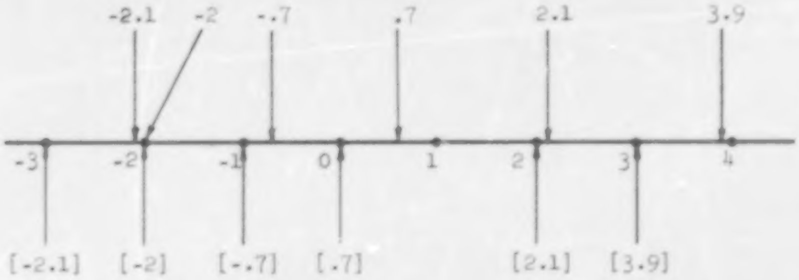


Figure 2-20

[†] No mathematician has ever seen the name GRIN. We've just made this up for our own convenience. The other function names you will see in this section like FRPT, TRUNK, and ROUNDUP are also just made up for our present purposes.

We see that for positive numbers the greatest integer function has the effect of "lopping-off" everything to the right of the decimal point. Whereas for negative numbers the instructions for finding the greatest integer would be: If anything but zeros appears after the decimal point, then lop off everything after the decimal point and subtract 1.

The graph of the greatest integer function (Figure 2-21) displays its step-like behavior.

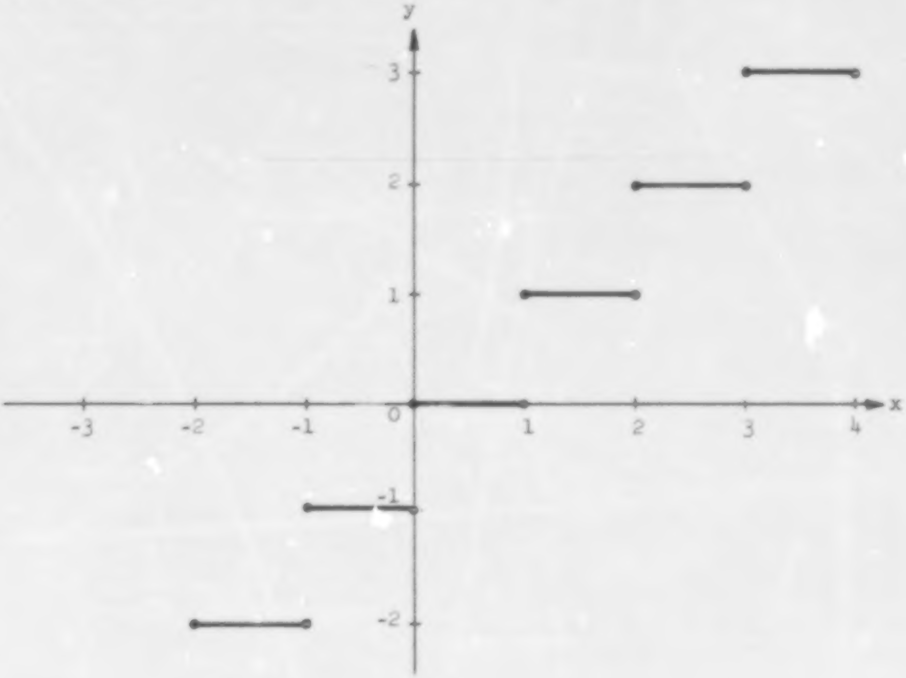


Figure 2-21. Graph of $y = [x]$

We also note that for all numbers x, we have $[x] \leq x$ or in other words, for all x,

$$x - [x] \geq 0 .$$

Let us use the notation FRPT(x) (for FRactional ParT of x) to denote $x - [x]$ and let us examine the graph of FRPT(x) given in Figure 2-22.

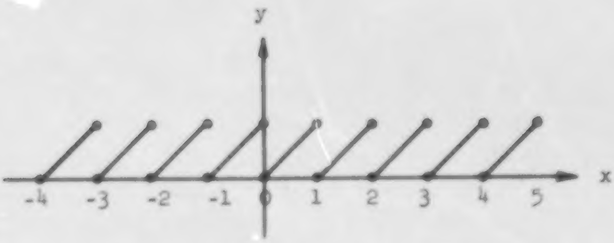


Figure 2-22. Graph of $y = \text{FRPT}(x)$

This graph exhibits the non-negativity of FRPT(x) as well as its property of being a periodic function of period 1. That is, we have $\text{FRPT}(1 + x) = \text{FRPT}(x)$ for all x. Geometrically, if we look, in Figure 2-22, at the functional value of a point x on the number line and then move one unit to the right, the new point will have the same functional value.

An interesting and important property of the functions $[x]$ and FRPT(x)--a property which could have been used in giving the definition of these functions--is that the equation

$$x = [x] + \text{FRPT}(x)$$

displays the unique decomposition of x as the sum of an integer and a non-negative number less than 1. Those students who have studied logarithms have been using these two functions whether they have used the names or not. If x is the logarithm of some number, then $[x]$ is its characteristic and FRPT(x) is its mantissa.

In Figure 2-23 the decomposition of the identity function, $\text{IDENT}(x) = x$ (dotted), as the sum of the GRIN(x) (solid) and FRPT(x) (dashed) is illustrated. At each point x on the x-axis the value of IDENT(x) is found by adding the values of GRIN(x) and FRPT(x).

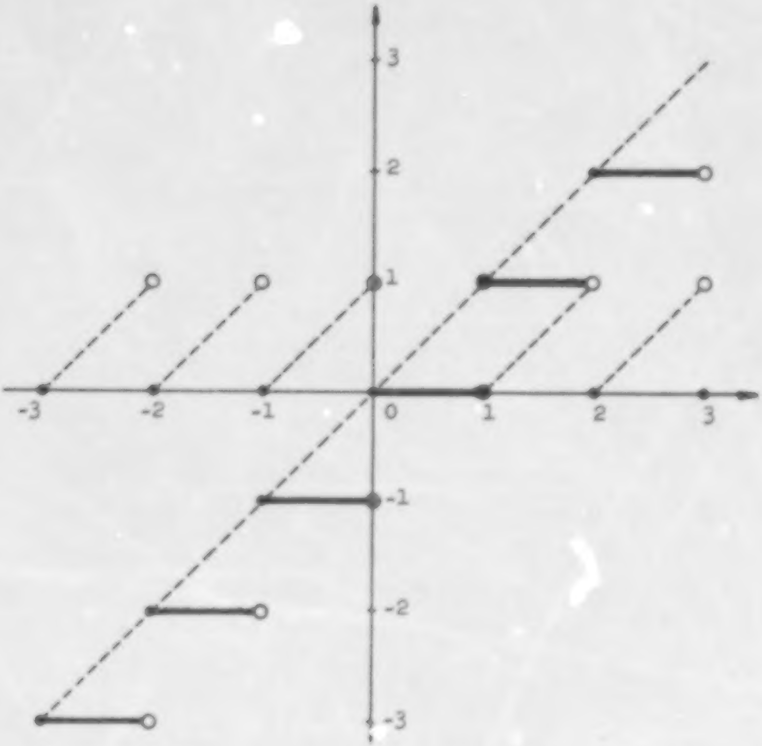


Figure 2-23

In many mathematical problems[†] we are interested only in the remainder obtained on dividing one integer by another. For example, if we divide 32417 by 1309 according to the rules learned in elementary school, our work looks like this:

$$\begin{array}{r} 24 \\ 1309\,\overline{)\,32417} \\ \underline{2618} \\ 6237 \\ \underline{5236} \\ 1001 \end{array}$$

We identify the numbers appearing here as:

dividend	32417
divisor	1309
quotient	24
remainder	1001

[†] e.g., See Euclidean algorithm in Section 3-2.

and we write

$$32417 = 1309 \cdot 24 + 1001 \quad .$$

In general if N is the dividend, M the divisor, Q the quotient and R the remainder, then

$$N = M \cdot Q + R \quad \text{where} \quad R < M \; .$$

This is the "division algorithm" of elementary school arithmetic. The values of Q and R are related to the functions studied in this section by

$$Q = \mathrm{GRIN}(N/M) \; ,$$
$$R = M \cdot \mathrm{FRPT}(N/M) \; .$$

These remainders are of fundamental importance in "modular arithmetic" where we replace all numbers by their remainders relative to some fixed divisor. In telling time in hours we use modular arithmetic modulo 12. In the "casting out nines" method of checking arithmetic we use arithmetic modulo 9. In the carnival wheel problems at the end of this section we encounter modular arithmetic modulo 4 and 5.

Just about any integer rounding function of practical value, i.e., related to interesting computer algorithms, can be expressed in terms of the greatest integer function. These relationships will be discussed later in this section. For the moment we will further illustrate the use of the $[x]$ function with the following problem.

> A farmer in a moment of weakness made a pledge which he now deeply regrets. The pledge was that he would keep all his money in multiples of $20.00 and that if any time he had a residue which was less than $20.00 this would be put into an educational fund for his son.

Thus in any monetary transaction in which the farmer receives money, only the number of twenty dollar bills he receives is of importance, while if he spends money the number of twenty dollar bills he must break is the important thing.

Suppose he sells the family cow for $75.75. How many twenty dollar bills will he receive? First we compute

$$\frac{75.75}{20.00} \quad \text{or} \quad 3.7875$$

and then we see that the number of twenties is 3.

Suppose he buys a horse for $87.50, how many twenties will he lose? First we compute (considering expenditures as negative)

$$\frac{-87.50}{20.00} \quad \text{or} \quad -4.375$$

and we see that the number of twenties he must break out is 5.

For either case it should now be clear that in any transaction the number of twenties is given by

$$\left[\frac{\text{AMOUNT}}{20.00}\right]$$

The amount going into the education fund will be given by

$$20 \times \text{FRPT}\left(\frac{\text{AMOUNT}}{20.00}\right)$$

Exercises 2-5 Set A

1. A small country in Europe has purchased 160,000 tons of grain from the U.S. It prefers to ship the grain in its only grain ship which has a capacity of 30,000 tons. What is the minimum number of round trips required?

2. We now generalize the problem in Exercise 1 as follows. Suppose this country buys grain rather frequently and in varying quantities. Further, suppose it has many ships at its disposal of various capacities, but only one ship is available at any one time which can be earmarked for handling the grain transport.

   Let TONS be the amount of grain purchased and CAPACITY be the capacity of the ship that is available. An algorithm to determine the number of round trips required is given in Figure 2-24.

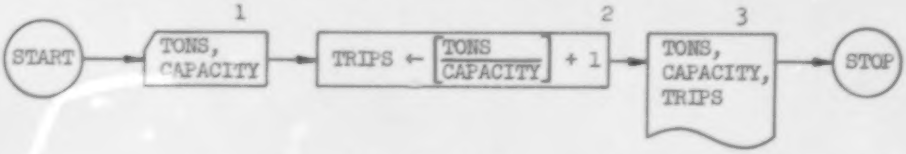

1
2
3
START
TONS,
CAPACITY
TRIPS ← $\left[\frac{\text{TONS}}{\text{CAPACITY}}\right] + 1$
TONS,
CAPACITY,
TRIPS
STOP

Figure 2-24. At Sea

Is this algorithm "seaworthy"? That is, assuming no error in the data, can this algorithm ever produce the wrong answer?

Approximation by Round-off

From the mathematical point of view, the expression

$$\frac{1}{3}$$

is a constant--a name for a particular number. From the computer point of view, this expression denotes a command--an indicated division which is to be carried out. If the machine encounters it in such a box as

$$\longrightarrow \boxed{x \leftarrow \frac{1}{3}} \longrightarrow$$

the net result will be that not $\frac{1}{3}$ but .333 or .33333 or .333333333 will be read into the storage location belonging to the variable x. The number of 3's which will be stored will depend on the number of digits the computer has been told to carry out in the divide operation or on the capacity of the storage location. In any case only finitely many digits of the infinite decimal representing $\frac{1}{3}$ can be stored so the computer replaces $\frac{1}{3}$ by some approximation; that is, it "rounds-off" the infinite decimal. Approximation by round-off is encountered almost every time we divide or evaluate a function as well as in many other places. In Chapter 6 we will say more about the effect that round-off has in computation and also discuss other interesting sources of "numerical error".

Although we have only been discussing integer rounding here, the rounding of any real number to the k^{th} place (where k is any integer) is a process which has as its heart integer rounding. Thus, if we wish to round a certain real number to the thousandths place $(k = 3)$, we can adopt the procedure:

1. Multiply the given number by 1000;
2. Round the result to an integer, employing some integer rounding function;
3. Divide the resulting number by 1000.

Example:

Round 57.29416 to the thousandths place.

1.	Multiply by 1000	57294.16
2.	Round to an integer (using the greatest integer function)	57294
3.	Divide by 1000	57.294 .

We are now ready for a precise definition:

An integer rounding procedure is a systematic method for replacing any given number by an integer subject to the conditions:

1. If the given number is an integer it is unchanged;
2. If the given number is not an integer then the rounded value is either the nearest integer to the left or the nearest to the right;
3. If $A \leq B$ then their rounded values satisfy the same inequality.

In more mathematical language the above can be stated:

> An integer rounding procedure is a monotone integer valued function, F, on the reals satisfying for all real x, the inequality, $|F(x) - x| < 1$.

Everything is found in the second definition that appears in the first except the word systematic, and we are unable to say what that means anyhow.

We see that when rounding a number which is not an integer we always have two choices for the rounded value. We can classify the four commonest rounding procedures according to how we make that choice.

1. Choose the first integer to the left;
2. Choose the first integer to the right;
3. Choose the first integer nearer to the origin;
4. [†] Choose the nearest integer.

The functions giving the rounded values are all closely related to $[x]$. These functions are given below. We leave to the student the simple task of checking that these functions actually do what we say they do.

1. Here the function is just $[x]$ itself;
2. $\mathrm{ROUNDUP}(x) = -[-x]$;
3. [††] $\mathrm{TRUNK}(x) = \mathrm{SIGN}(x) \times [|x|]$;
4. $\mathrm{ROUND}(x) = [x + .5]$.

[†] Note the ambiguity for numbers halfway between two integers. We will select a simple formula which works for other numbers and take what it gives for odd multiples of 1/2.

[††] Here $\mathrm{SIGN}(x)$ is defined to be $x/|x|$ unless $x = 0$ in which case $\mathrm{SIGN}(x) = 0$.

The method of rounding given by TRUNK is called truncation. It can also be described as "lopping-off" everything after the decimal point, regardless of sign. Notice that for positive x, TRUNK(x) and GRIN(x) are the same.

The TRUNK function is employed implicitly in programming languages like FORTRAN and MAD when converting real numbers to integers. The ROUND function plays an equally prominent role in ALGOL implementations when converting real numbers to integers. The ROUNDUP function is not important but is included in these discussions to round out our discussion of rounding.

When calling for the division of two integers I and J we quite often really want TRUNK(I/J) which is the integral portion of the actual quotient, I/J. Many programming languages have special conventions that enable us to imply TRUNK(I/J) without having to bother to write TRUNK. When the quotient of I and J is indicated in such a way that it means TRUNK(I/J) for example, by writing "I ÷ J" (as in ALGOL), we call this integer division.

We usually think of these rounding procedures as producing approximate answers to problems. However, in problems which by their very nature require whole number answers, it sometimes happens that these rounding procedures are tailor-made for producing the exact answers required. Such situations are emphasized in the following exercises.

Exercises 2-5 Set B

1. It costs 8¢ an ounce to send an airmail letter. Write a formula involving one of the rounding functions expressing the cost of sending an air mail letter as a function of the (real) variable WT.

2. A camp director wishes to divide the boys into baseball teams. Give a formula involving one of the rounding functions giving the number of teams as a function of NBOY (the number of boys). No boy is to be on more than one team.

In each of the following three exercises, your job is to plot a graph similar to Figure 2-21.

3. For ROUNDUP(x) from $-3 \leq x \leq 3$.
4. For TRUNK(x) from $-3 \leq x \leq 3$.
5. For ROUND(x) from $-3 \leq x \leq 3$.

6. Graph the following four functions on one set of axes. Be sure to limit the domain of each of the functions according to the inequality that accompanies the function.

(1) $y = [x]$ for $-3 \leq x \leq 3$

(2) $y = [x] + 1$ for $-3 \leq x < 3$

(3) $x = [y]$ for $-3 \leq y \leq 3$

(4) $x = [y] + 1$ for $-3 \leq y < 3$

7. Determine the values of ROUND(x) when x is an odd multiple of $\frac{1}{2}$.

8. A game wheel is divided into five equal sectors numbered consecutively from 1 in a clockwise manner as shown in Figure 2-25. There is a spinner which rotates on a shaft mounted at the center of the wheel.

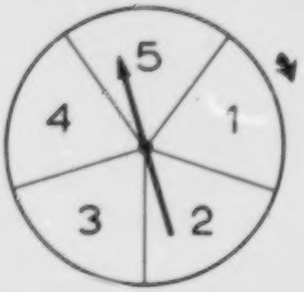


Figure 2-25.

Let S be the sector pointed to by the spinner at rest. We now flick the spinner with our fingers in a clockwise direction. It spins through m sectors and comes to rest inside a sector, i.e., not on a line.

(a) Write a formula involving one of the rounding functions which gives you the new sector number NEWS in terms of the original rest position S and the spin span M.

(b) What changes are needed in the formula found in (a) to make it applicable for spins in either the clockwise or the counterclockwise directions?

(c) Generalize the formula(s) developed previously in this problem to the case of a game wheel having k sectors numbered consecutively from 1.

9. A carnival wheel, Figure 2-26, has 32 painted sectors numbered clockwise, $s = 0, 1, 2, \ldots, 31$. The sectors are divided into 8 groups, 4 sectors per group. In each group, the sectors are painted blue, green, red and yellow (B, G, R, and Y) going clockwise.

When the wheel is spun (always counterclockwise) and comes to rest, the color of the sector opposite the fixed pointer, R, tells you how the game comes out.

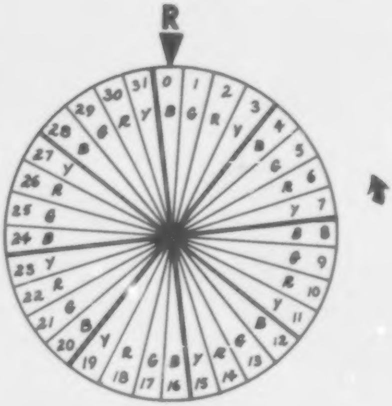


Figure 2-26

Suppose the rule is

Player loses 30 points for blue.
Player loses 10 points for green.
Player wins 10 points for red.
Player wins 30 points for yellow.

Further suppose that, before any one spin the wheel is considered to be at rest with sector number s opposite the ratchet R. We now imagine the wheel is spun a distance of m sector positions. How many points p will be won or lost for each data pair s and m? How can we develop a simple algorithm which simulates repeated plays at the wheel?

HINT: Your flow chart should show a loop beginning with a step for the input of s and m, one or more assignment boxes to compute p, an output statement to print p, and a return to the input step. One way to compute p, is to first compute the new sector number s after the spin, in terms of the given (or old) s and m. Then we can compute the position k (= 0, 1, 2, or 3) within the group--corresponding to blue, green, red or yellow, respectively. (Actually it is simpler to compute k directly from m and the old s without first computing the new s.) To simulate repeated spins, return to the input step after printing p.

2-6 Alphanumeric data[†]

A funny thing happened one day when the master computer sent his robot the "reader" to a window box. The robot returned, in tears and consternation, with this story--"You sent me to the window box marked X to bring you its value. When I looked through the window to copy the number, there was no number--only the letter 'X'.

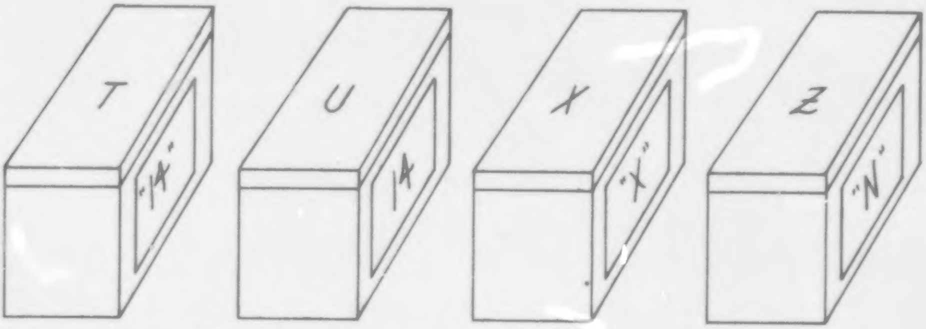


Figure 2-27. "... but when I opened the box to copy the number, there was no number--only the letter 'X'."

Then I went to the box marked Z and, to my horror again I found not a number but a letter. This time it was the letter 'N'. Please, sir, what does this mean?"

Reading this story may make you as confused as the poor robot. We hope not. There are great rewards for those who will grasp its true meaning. We learned in the preceding section that computers can read and store alphabetic characters and special characters in the words of memory. Coupled with what we can recall from Section 1-4, it seems that characters like "1", "4", "7", or like "T", "X", "N", or like "*", "/" and ")" can each be stored one or more per word of memory as a special combination of six bits. So it's entirely possible for a string of characters, say, "14", to appear in memory as an entirely different pattern of bits than, say, the integer 14. If we imagine a memory word of twelve bits, then following Figure 1-14, "14" would be coded as 000001000100 while the integer 14 would be coded as 000000001110. If one makes the mistake of misinterpreting 000001000100 as an integer one would then read it as 68.

[†] Whenever you see a daggered section heading, you can assume the material is interesting, but, if time is short, the whole section can be skipped without loss of continuity, especially during a first reading.

If you find a daggered paragraph somewhere in the middle of a section, it means it's possible to skip to the end of the section.

If you see two daggers ††, it means the material may be even more interesting, but even more reason to skip it if you are pressed for time.

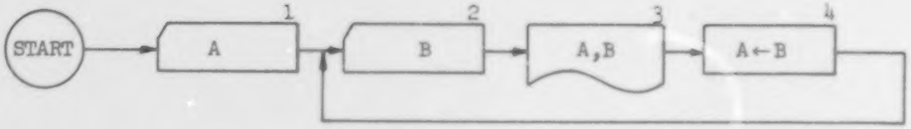
How can these facts relate to our flow chart language? Well, for one thing we should be able to see that a window box can store characters as well as numbers. In other words, a variable X can have a value that is not numerical at all but alphanumerical. By alphanumerical we shall mean a value consisting of some collection of characters made up of those displayed on the card in Figure 1-15.

Just how many characters can be stored in one window box depends on the size of the box--or memory word size. For this text, since we aren't dealing with any one computer, we won't be too specific. Let's assume that a window box can store a string of "several" characters. We will leave it to your language manual or your laboratory instructor to be more specific on this point.

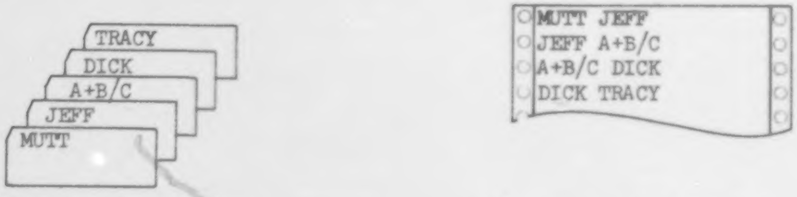
If a variable can have an alphanumeric value, it must be able to acquire such a value the same way it can acquire a numerical value, namely as a result of input or as a result of an assignment step. Having once acquired an alphanumeric value, it must be possible to output it by an output step.

It begins to appear that our input, output and assignment boxes must allow us to describe computer procedures for doing things with alphanumerical as well as numerical data!

We immediately illustrate this point by showing a very simple flow chart (Figure 2-28), the input data consisting of names, one per card, and the printed results--a list of name pairs.



(a) The flow chart



(b) The data

(c) The printed results

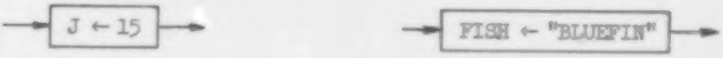
Figure 2-28. A questionable process

What do you imagine is in the window box called A before and after Box 4 is executed for the first time? To answer this, perhaps we had better step through the process once from the very beginning.

When Box 1 is executed, the four letters, "MUTT", are read from the card and are assigned to the variable A. Now Box 2 is executed where "JEFF" is assigned to B. Then values of A and B are printed at Box 3. When we come to Box 4 we see that at first A has the value "MUTT", but after Box 4 is executed, the current value of B which is "JEFF" will have been assigned to A. In answer to our original question, A has the value "MUTT" before the first execution of Box 4 and the value "JEFF" after. If you're wondering about the third card in the stack, it got in there by mistake, but we deliberately left it in to illustrate how our algorithm takes it in stride.

Observe how we have been using quote symbols to describe alphanumerical values. We don't actually put them on the data card, as you can see in Figure 2-28(b), and they don't actually appear when printing the alphanumerical values either--as you can see in Figure 2-28(c). Moreover, we also avoid quote marks around variables, like A.

As you will see presently we can have alphanumerical constants and we can assign such constants to variables. The parallel is illustrated in Figure 2-29.



(a) assignment of a numerical constant

(b) assignment of an alphanumerical constant

Figure 2-29. Two kinds of assignment

Example (a) shows a "conventional" assignment of a constant value to a variable. Example (b) shows an assignment of the character string "BLUEFIN" to the variable FISH. Any quantity in quotes is to be regarded as an alphanumerical constant.

We have come to the end of our parallel. More complicated expressions to the right of the arrow will be considered meaningless and will not be permitted.

For example

FISH ← 2 + "BLUEFIN"
or FISH ← 2 × "BLUEFIN"
or FISH ← "BLUEFIN" + "REDFIN"

are, as far as we are concerned, meaningless. We see that there are only two allowed forms of alphanumeric assignment steps:

variable ← variable

and

variable ← alphanumerical constant .

It should now be clear, in spite of all temptation, that the following are also invalid forms:

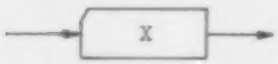
NUMBER ← 2 + "4"

or

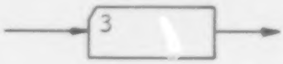
PRODUCT ← "5" × "5" .

One more crucial observation must be made here.

Suppose that in carrying out the input step,



a data card like the following arrives in position to be read.



How do we specify in our flow chart language whether it is to be read as a number or as a symbol? The answer is: if any box in the flow chart contains an operation on the data which can be performed only on numbers, then the value on the card *must* be read as a number, but if there are no such operations then you may choose *either way* to read it. However, before the card is read this decision must have been faced and made. There is no ambiguity when the card arrives in position to be read.

Now let us look at some examples of input and output of symbols and numbers to illustrate this thought. Let's first imagine we have a flow chart, Figure 2-30, for the input of two values, X and Y and the output of their sum, Z. Two different data cards are presented for input as shown in Figure 2-31. If the first card is read, everything works fine. The answer, for input values of 4 and 3 is 7.

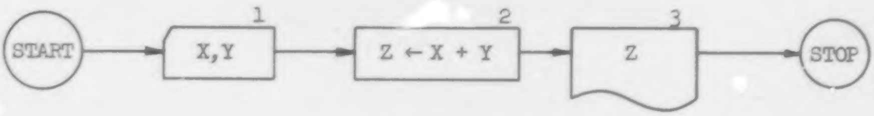


Figure 2-30. First flow chart

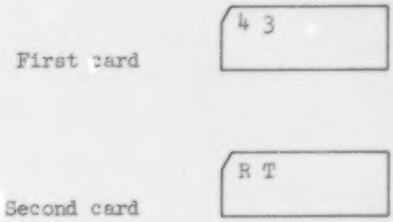


Figure 2-31. Two data cards

If the second card is read, what happens? Something is obviously wrong because we cannot add "R" to "T". A perfectly valid flow chart when used with data which can be interpreted as numerical becomes utterly meaningless for data that is clearly not numerical.

Now let's look at a second flow chart (Figure 2-32), which inputs two values X and Y, assigns Y to Z and then prints the values of X, Y and Z. If we present the first card as input, there is no problem. The computer prints three values, "4", "3", and "3". If we present the second card as input, no problem. The computer prints "R", "T", and "T".

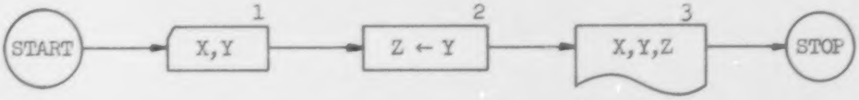
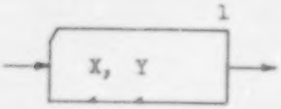


Figure 2-32. Second flow chart

Here then is a flow chart which can be said to be meaningful whether the data is either numerical (or may be interpreted as numerical) or alphanumerical.

You can look at the first flow chart and pretty clearly say that it is intended for work on numerical values only--i.e., that the window boxes for X, Y and Z are expected to store only numbers. Box 2 tips you off to this crucial fact. But if you look at the flow chart in Figure 2-32, you simply cannot say what types of values the window boxes should or should not be allowed to have stored in them.

We see, then, that the flow chart alone will not always make crystal clear what kinds of data are to be assigned to each of the variables. If you feel there is an intolerable ambiguity creeping in here, we can simply agree to flag (in our flow chart) those input variables that are to be treated as alphanumeric. For instance in Figure 2-32 we can revise Box 1 as



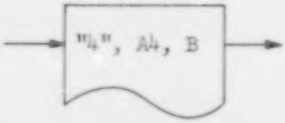
putting a little "notch" under each variable of the input list whose input value is to be treated as alphanumerical.

On the other hand, you may be willing to live with this situation because

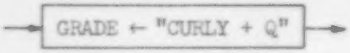
(a) in this case (Figure 2-32) it simply doesn't matter, and

(b) in an actual computer programming language like FORTRAN or ALGOL, simple steps are always taken to remove such ambiguities. You will see this when you consult your language manual.

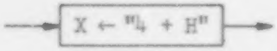
In any event, remember to use quote marks around numerals or character groups only when you mean them to be character groups, as in



or



or



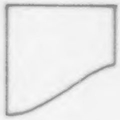
and avoid them otherwise.

Chapter 3

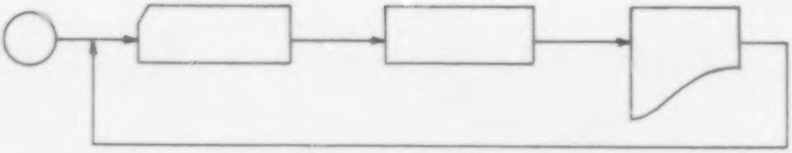
BRANCHING AND SUBSCRIPTED VARIABLES

3-1 Branching

So far our flow chart tools and techniques include input, assignment and output boxes



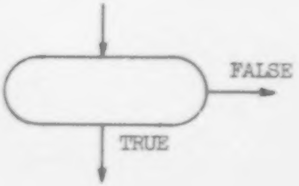
and the idea of a loop such as:

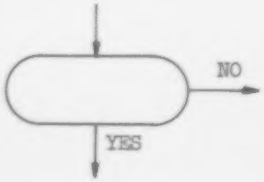


With these tools we have seen how we may make flow charts for algorithms which call for many repetitions of the same calculations with different sets of data.

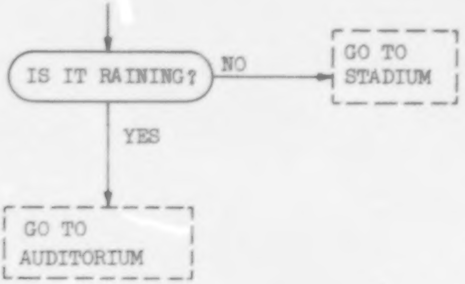
In this chapter we add two new tools to our kit. Used in combination with those we already have, these new tools enable us to construct flow charts for algorithms of any degree of complexity. The first of these is "branching" which gives us the ability to choose a new path (or branch) depending on whether a certain condition is satisfied.

Branching is indicated in flow charts by a "condition box," oval in shape.





As you can see, a condition box has two exits. In this way a condition box differs from all the other boxes we have met in flow-charting. Here is a non-mathematical analog of the use of the condition box. The University president has announced that graduation ceremonies will take place in the stadium unless it rains, in which case they will be moved to the auditorium. A flow chart of our behavior in attending the graduation would include



To illustrate the use of branching in computation we offer the following story book problem.

Example: The Ruritanian Post Office Department has just announced the regulation that no packages will be accepted for mailing which are greater than 29 inches in diameter. [By the diameter of a package the Ruritanians mean the maximum of the distances between pairs of points in the package. For a rectangular box the diameter is the length of the interior diagonal.] A wholesaler with a large number of boxes packed and ready for mailing must now see which packages comply with the new regulation and which will have to be repacked. He has no way of directly measuring the diameter but he can measure the three edges.

A computer programmer tells the wholesaler to write an identifying number on each package and to prepare for each package a Hollerith card. The card is to have punched in it the identifying number (N), and the lengths of the three edges (A, B, C). The computer will then be instructed to read these cards and to print a list of the numbers of the packages which comply with regulations. Here is the flow chart.

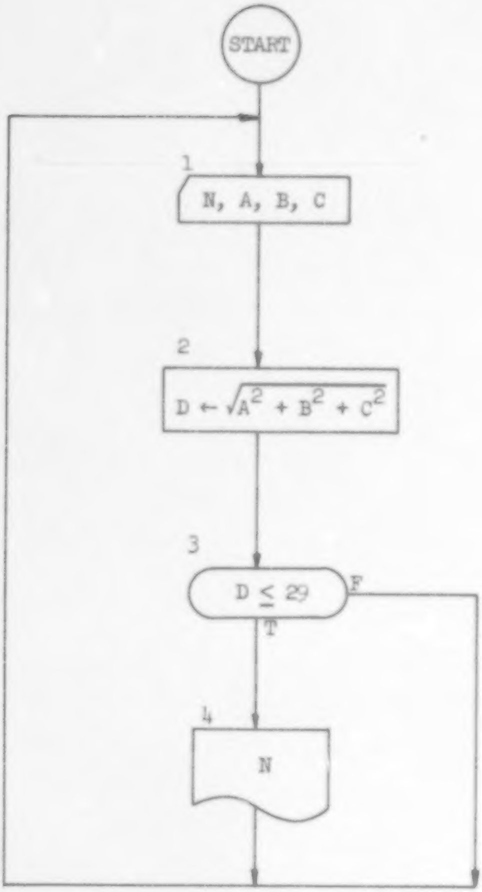
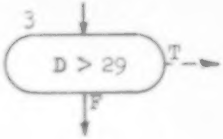
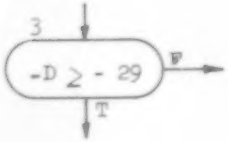

START
1
N, A, B, C
2
D ← √(A² + B² + C²)
3
D ≤ 29
F
T
4
N

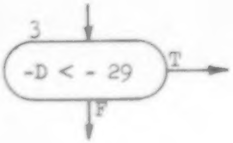
Figure 3-1. Flow chart for postal regulations

This is easily recognized as a modification of either Figure 2-11 or 2-6. You should check that the desired result is obtained, a list of the packages complying with the postal regulations.

It will be of interest to know that we could, if required, replace the condition box 3 by any of the following.



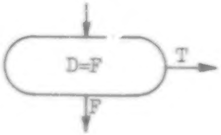




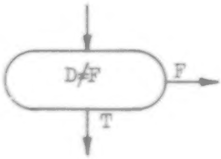
By saying we can replace one box by another we mean that entering each of these boxes with the same value of D we will always come out in the same direction. This is another thing for the student to check.

We see, then, that it would be possible to restrict ourselves to the use of just one of the four inequalities $\leq$, $<$, $\geq$, $>$. But we do not make this restriction. We write our inequalities in condition boxes in whatever form comes most naturally to us.

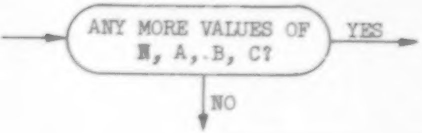
A choice is also available when two values are compared by employing the " = " and " ≠ " symbols. For example,



or alternatively,



In our relatively informal flow chart language we permit just about any form of question or assertion in our condition boxes. As an example we could insert the condition box


ANY MORE VALUES OF N, A, B, C?
YES
NO

into the last flow chart to obtain

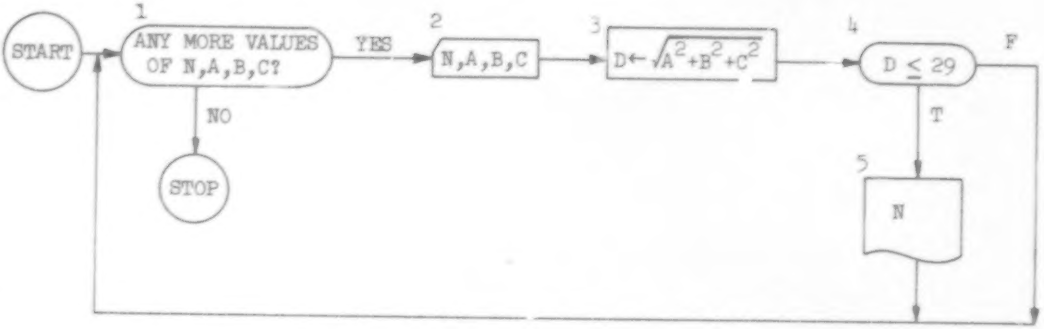
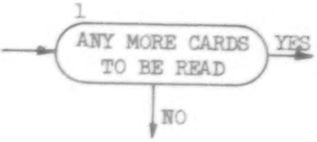
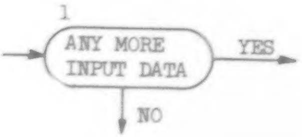

START
1
ANY MORE VALUES OF N,A,B,C?
YES
NO
STOP
2
N,A,B,C
3
D← √A²+B²+C²
4
D ≤ 29
F
T
5
N

Figure 3-2. The Ruritanian problem solved

In our informal flow chart language we could as well have written for box 1:

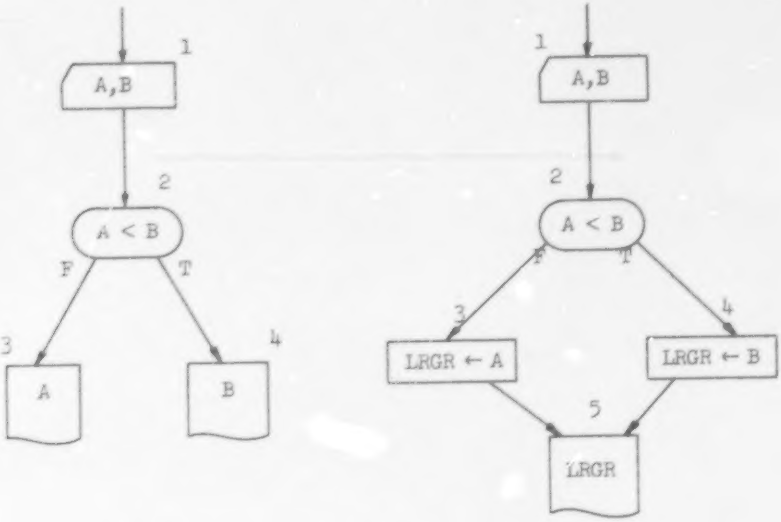

1
ANY MORE CARDS TO BE READ
YES
NO

or


1
ANY MORE INPUT DATA
YES
NO

Box 1 is not a logical necessity in Figure 3-2 since we adopted the convention [illegible] 2 that an input box was constructed so as to stop the computation [illegible] nothing is left for input. Nevertheless, it is good practice to include such a box and we shall usually do so. Some reasons for doing so are: not everyone follows the above-mentioned convention; if we wish to use the calculation of Figure 3-2 as a part of a larger algorithm, we are all ready to branch to another task rather than stopping; explicitly exhibiting the command to stop makes it easier to avoid "endless loops."

One obvious example of the use of decision is in determining and printing the larger value of a pair of numbers. Here are two ways of doing this:



(a) First Form

(b) Second Form

Figure 3-3. Two flow charts for larger

There is a valuable lesson to be learned in Figure 3-3. Several quite different flow charts may represent the same problem. Each of these flow charts may have both advantages and disadvantages relative to the others. One advantage to the first form in Figure 3-3 is that it is possible to determine which variable has the larger value. By "larger" here we really mean greater than or equal to. Figure 3-4 illustrates this idea.

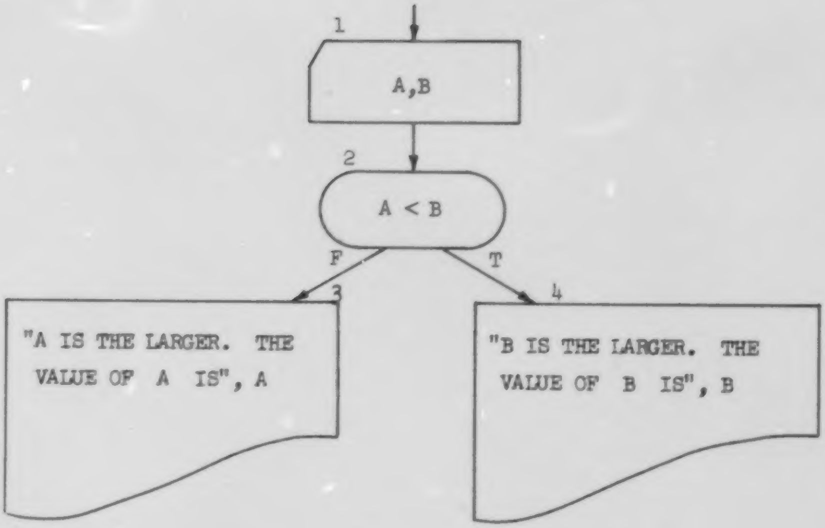
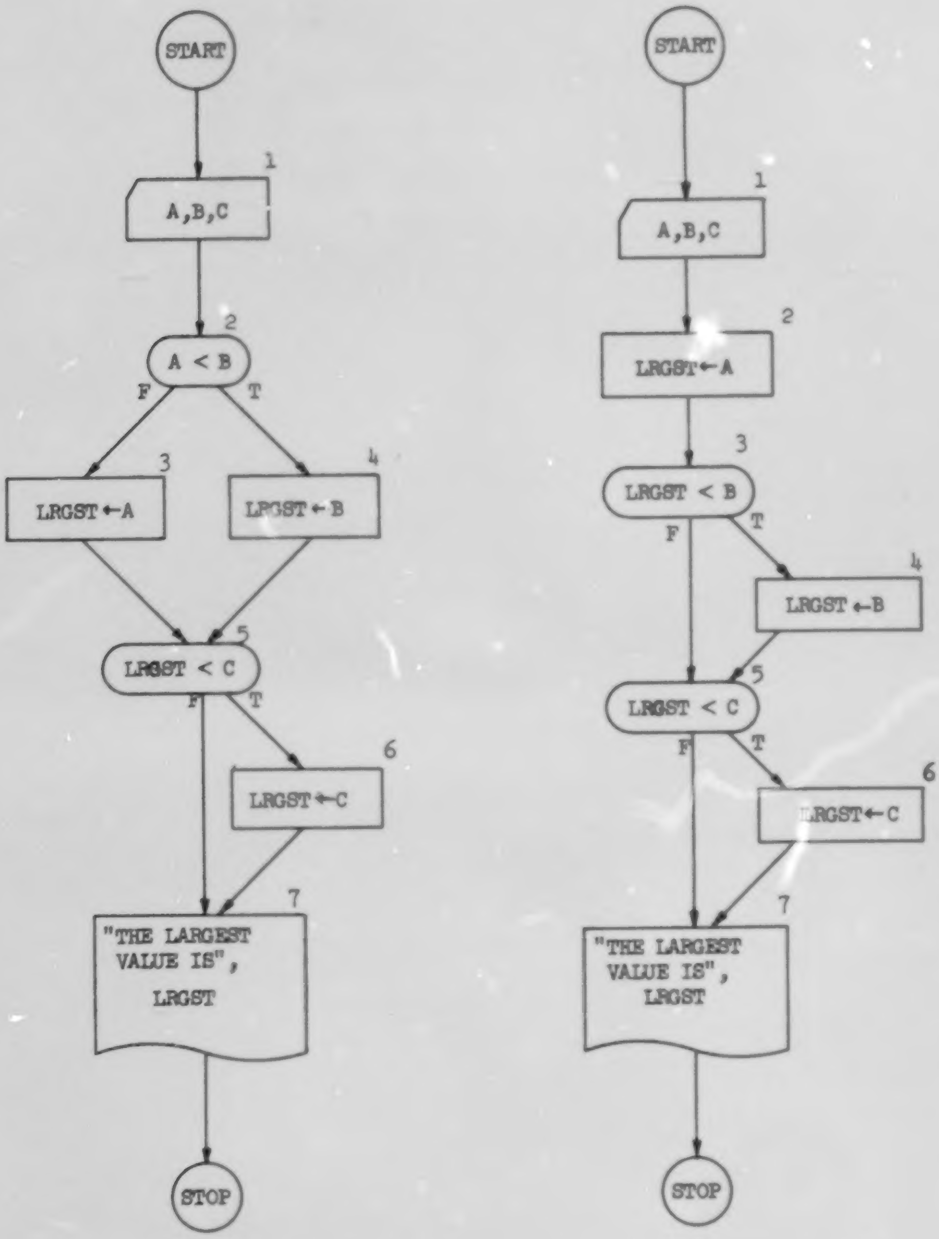

1
A,B
2
A < B
F
T
3
4
"A IS THE LARGER. THE VALUE OF A IS", A
"B IS THE LARGER. THE VALUE OF B IS", B

Figure 3-4. Output with identifying remarks

In the output boxes 3 and 4, anything enclosed in quotation marks is to be printed just as it appears. Variables not enclosed in quotes will have their current values printed. Individual output items are to be separated by commas.

An advantage to the second form in Figure 3-3 lies in the ease with which we can generalize it to more variables. We give two flow charts exhibiting this generalization.


START
1
A,B,C
2
A < B
F
T
3
LRGST ← A
4
LRGST ← B
5
LRGST < C
F
T
6
LRGST ← C
7
"THE LARGEST VALUE IS", LRGST
STOP
START
1
A,B,C
2
LRGST ← A
3
LRGST < B
F
T
4
LRGST ← B
5
LRGST < C
F
T
6
LRGST ← C
7
"THE LARGEST VALUE IS", LRGST
STOP

(a) First Form

(b) Second Form

Figure 3-5. Two flow charts for selecting the largest of three values

The second form of Figure 3-5 has an advantage over the first form which we cannot fully appreciate at this point. This advantage lies in the uniform format of the comparisons. We will understand the significance when we study subscripted variables later in the chapter.

A flow chart for the same problem which generalizes the first form in Figure 3-3 is

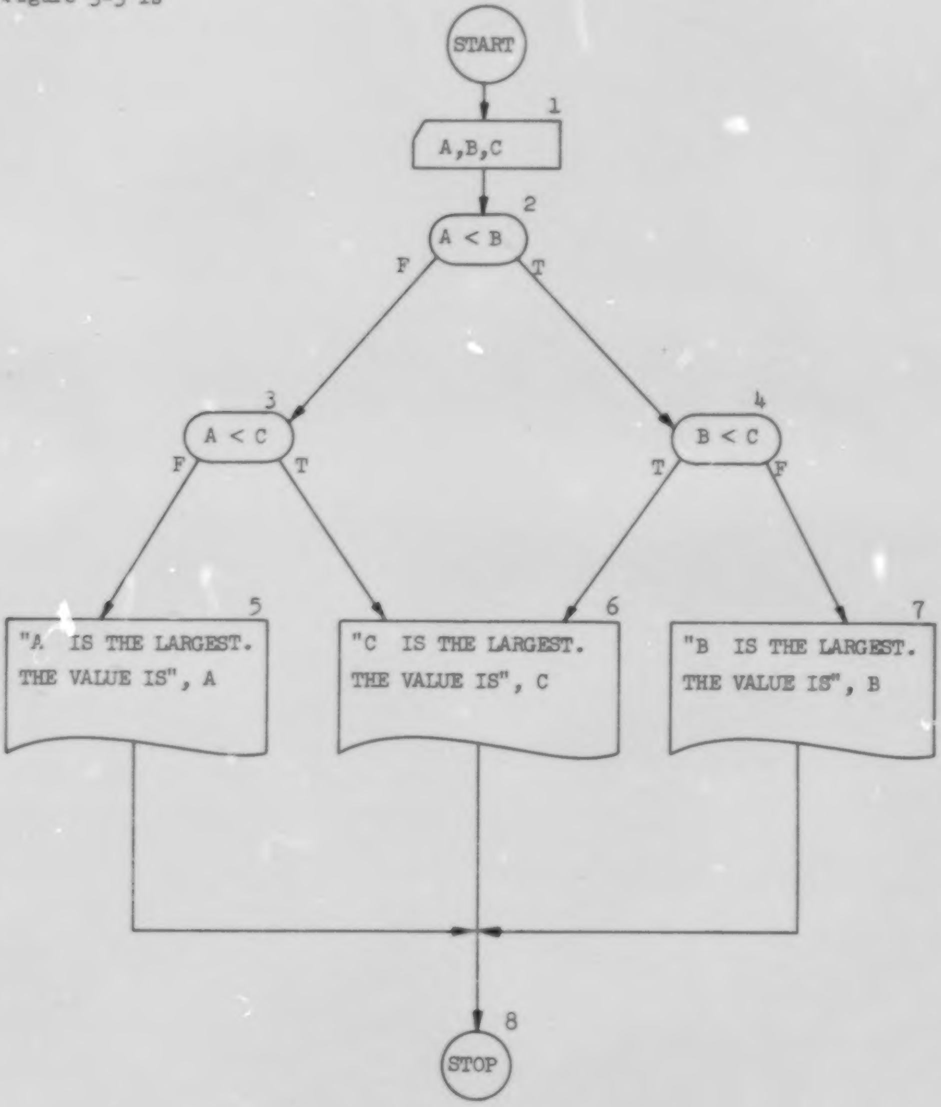

START
1
A,B,C
2
A < B
F
T
3
A < C
4
B < C
F
T
T
F
5
"A IS THE LARGEST. THE VALUE IS", A
6
"C IS THE LARGEST. THE VALUE IS", C
7
"B IS THE LARGEST. THE VALUE IS", B
8
STOP

Figure 3-6. Flow chart for "largest" without use of auxiliary variables

The next flow chart--the last of this section--shows how a computer may be used to tally data read from n cards. Here the input variable, T, represents test scores punched on cards. It is desired to know how many scores fell in the low range $(0 \leq T \leq 50)$, how many in the middle range $(50 < T \leq 80)$ and how many in the high range $(80 < T \leq 100)$. The variables low, mid and high act as counters. For each input value of T one of the three counters clicks up one notch. The initial assignment box sets these counters to zero. Another counter called "count" keeps a tally of the number of data values read thus far. When count reaches n, which is input at Box 1, the printout at Box 11 is executed.

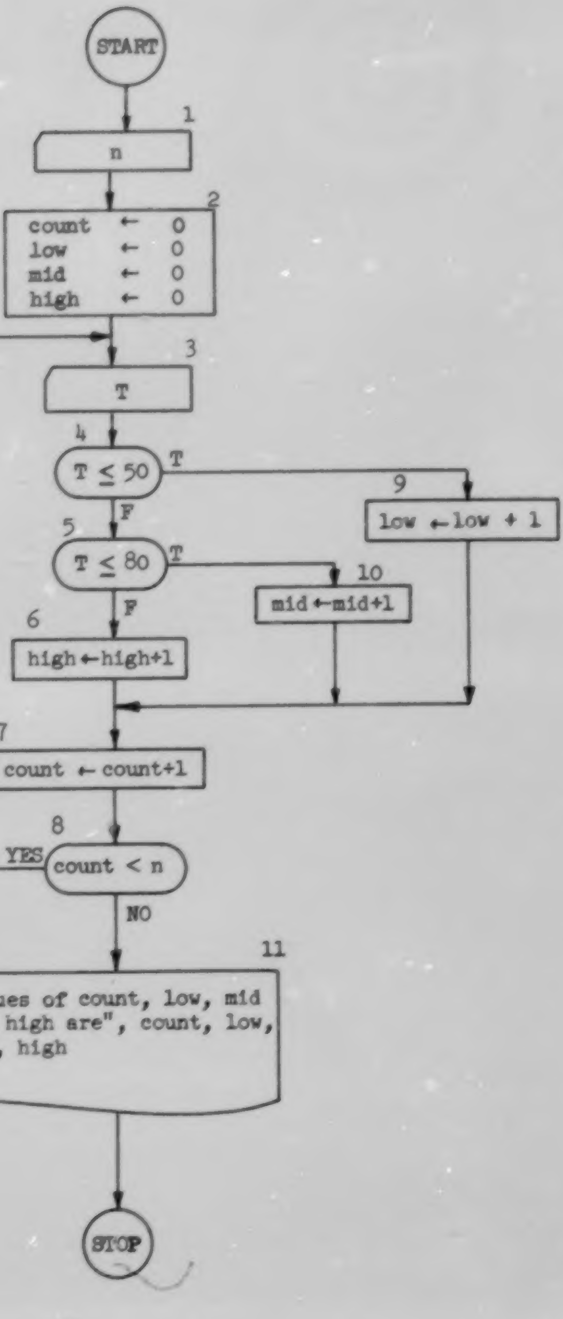

START
1
n
2
count ← 0
low ← 0
mid ← 0
high ← 0
3
T
4
T ≤ 50
T
F
9
low ← low + 1
5
T ≤ 80
T
F
10
mid ← mid+1
6
high ← high+1
7
count ← count+1
8
YES
count < n
NO
11
"Values of count, low, mid and high are", count, low, mid, high
STOP

Figure 3-7. Illustrating use of two million dollar computer for tallying

Exercises 3-1 Set A

In Exercises 1 - 3 use the flow chart in Figure 3-3(b) but replace (mentally) the condition in Box 2 by '$2A < B$'. Give as output the value of A or B selected by this criterion from the input indicated in the exercise.

1. $A = 5,\ B = 7$

2. $A = -3,\ B = -5$

3. $A = 10,\ B = 5$

4 - 6. Work Exercises 1 - 3, but use the condition '$A^2 - B \leq B^2$' in Box 2.

Draw flow charts for the processes specified in Exercises 7 - 9. Begin by reading as input values of b, c, d, and x, then print these values as output. Then complete the process indicated.

7. If b is greater than c, output the value of d. Otherwise, output the value of x.

8. If $d < c$, output the value of $c \times b + d \times x$. Otherwise, output the value of $d - c$. Hint: You cannot print the value of any expression like $c \times b + d \times x$ without first evaluating it and assigning it to a variable.

9. If $(b + c)^2 + x^2 > b \times c \times d$, output the value of $b + c + b \times c \times d$. Otherwise, output the value of $b^2 \times c^2 \times d^2$ and $(b + c)^8$.

10. Prepare a flow chart for an algorithm which inputs values of j, m, and n, determines the sum of j and the larger of m and n and outputs the values of j, m, n, and the sum. The last step is to return to the beginning to input more values of j, m, and n.

11. Draw a flow chart to input values of b and c, output both values immediately, and then perform the following:

    If $b = 0$ and $c \neq 0$ output "$bx + c = 0$ has no root."

    If $b = 0$ and $c = 0$ output "every real number satisfies $bx + c = 0$".

    If $b \neq 0$ compute the root of the equation $bx + c = 0$. Output "the root of $bx + c = 0$ is", followed by the root.

    Finally, return to the input step for more data.

Exercises 3-1 Set B

In the tallying problem, Figure 3-7, we saw how a computer might be asked to examine and tally a series of values for T that are input from data cards. There are many similar things we may want to do with a series of input values. For example, we may wish to sum all the values of T, or sum the squares of T, or sum the absolute values of T, etc. In the following exercises, develop a flow chart for the described operation on a series of input values for T. Always print some appropriate message which identifies the numerical result that is also to be printed. The basic ingredients for the desired flow charts can be found by re-studying Figure 3-7.

1. Sum 100 values of T and print out the sum. Call this sum SUMALL.
2. Sum the cubes of 100 values of T. Call this SUMCUB.
3. Sum only the negative values found in 100 input values for T. Call this SUMNEG.
4. Without reading the input values more than once, develop all three sums; SUMALL, SUMCUB, and SUMNEG.
5. For each of the 100 values that are input, print the cumulative sum to that point. Call it CUMSUM. Thus, after reading the 5th value for T, we print the sum of the first 5 values. After the 6th value of T has been read, we print the sum of the first 6 terms, etc.
6. Think of the hundred input values mentioned in the preceding exercises as representing the plays of a game which has two players. If a number is ≥ 0 it means player A has won that play. If the number is negative, player B has won that play. Now suppose the game is scored as follows (like badminton or volley ball): Player A begins by serving. If the server wins a play, a point is added to his score. If the server loses a play, the other player becomes server and the score does not change. Prepare a flow chart to print which player wins and the score after 100 plays.

3-2 Auxiliary Variables

In the previous section we saw the introduction of "auxiliary variables" into our flow charts. By auxiliary variables we mean variables not obviously involved in the phrasing of the problem. We used these variables only for convenience. In this section we will see some rather unexpected uses of auxiliary variables.

Consider the Fibonacci Sequence

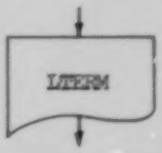
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

where we start with two 1's and then form new terms according to the rule that each term is the sum of its two predecessors. We will construct a flow chart for computing the first thousand terms of this sequence.

Instead of presenting you with the finished flow chart we will have a look at the process of its construction. What we want to do is to grind out the terms of the Fibonacci Sequence and to print the latest term as we compute it. For ease of recall let us introduce the variable

LTERM

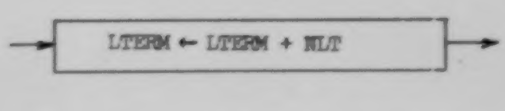
to represent the Latest TERM. We will then want to have in our flow chart a print instruction of the form:



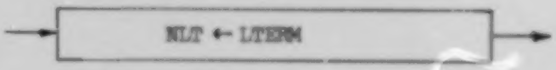
To compute the next value of LTERM we need to add the present value of LTERM to the value of the Next to Last Term which we call

NLT

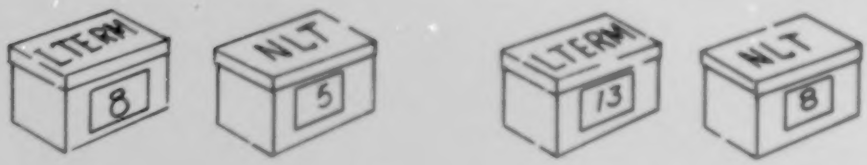
The fundamental step in this program will be the computation of the new value of LTERM and the assignment of this value to the variable LTERM. This step is indicated by



At the same time the previous value of LTERM gets demoted to second place, that is, to NLT. This is indicated by the box



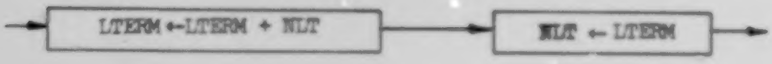
But which of these two assignment statements should come first? Let us test this by going back to our window boxes. Let us take the point in construction of the series at which LTERM has the value 8 so that NLT has the value 5. After all the switching is done LTERM should have the value 13 while NLT has the value 8, as suggested in Figure 3-8.



Before After

Figure 3-8. Desired effect of assignment statements in above discussion

First we will try



The effect of this is shown in Figure 3-9.

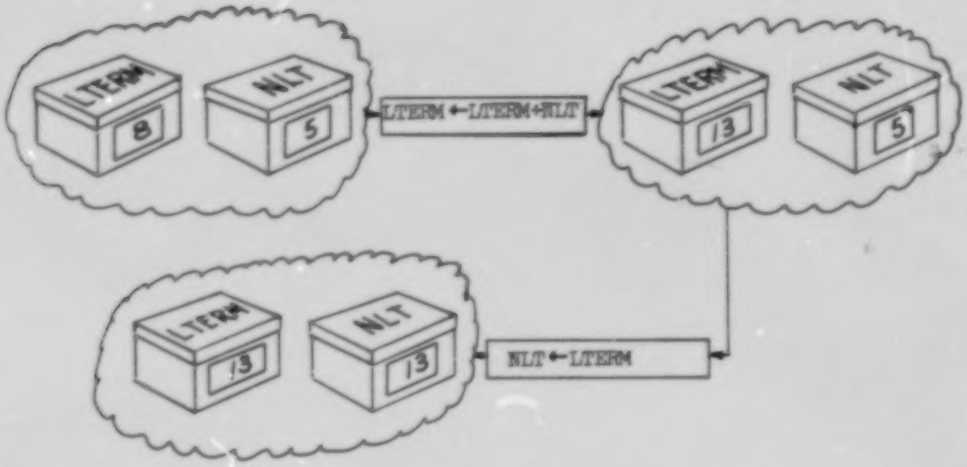

LTERM
NLT
8
5
LTERM ← LTERM+NLT
LTERM
NLT
13
5
NLT ← LTERM
LTERM
NLT
13
13

Figure 3-9. First effort to attain desired effect

No good! So, we will try it the other way around. (Figure 3-10.)

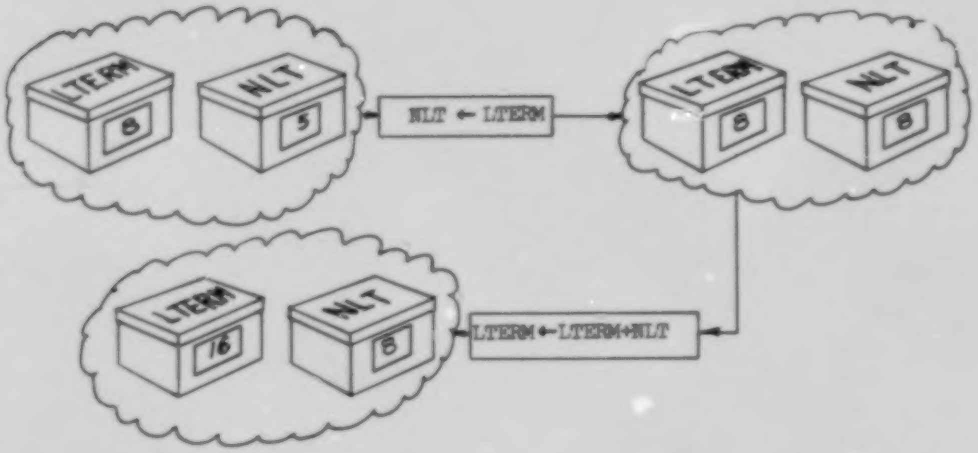

LTERM
NLT
8
5
NLT ← LTERM
LTERM
NLT
8
8
LTERM ← LTERM+NLT
LTERM
NLT
16
8

Figure 3-10. Second abortive effort

This is truly a shattering blow! Is there no way to write a flow chart for our intended algorithm for the Fibonacci Sequence?

It is important to subject our failure to analysis. What we were really thinking was somewhat as follows. Consider the two assignment statements

LTERM ← LTERM + NLT

NLT ← LTERM

First, evaluate the right hand sides of both with the original values of the variables, say 8 for LTERM and 5 for NLT. Next, simultaneously make the indicated assignments. The desired values are now evidently assigned to the variables. But an algorithm is a plan for carrying out a process in a finite number of steps and with a computer every step must be carried out in a definite sequence--not simultaneously. As soon as we assign a new value to a variable, the old value is lost forever--unless we have had the foresight to make a copy of it. Therein lies the solution to our dilemma. We introduce a new variable COPY, and consider, in order, the following assignment steps:

COPY ← LTERM

LTERM ← LTERM + NLT

NLT ← COPY

Following as before with window boxes yields the results shown in Figure 3-11.

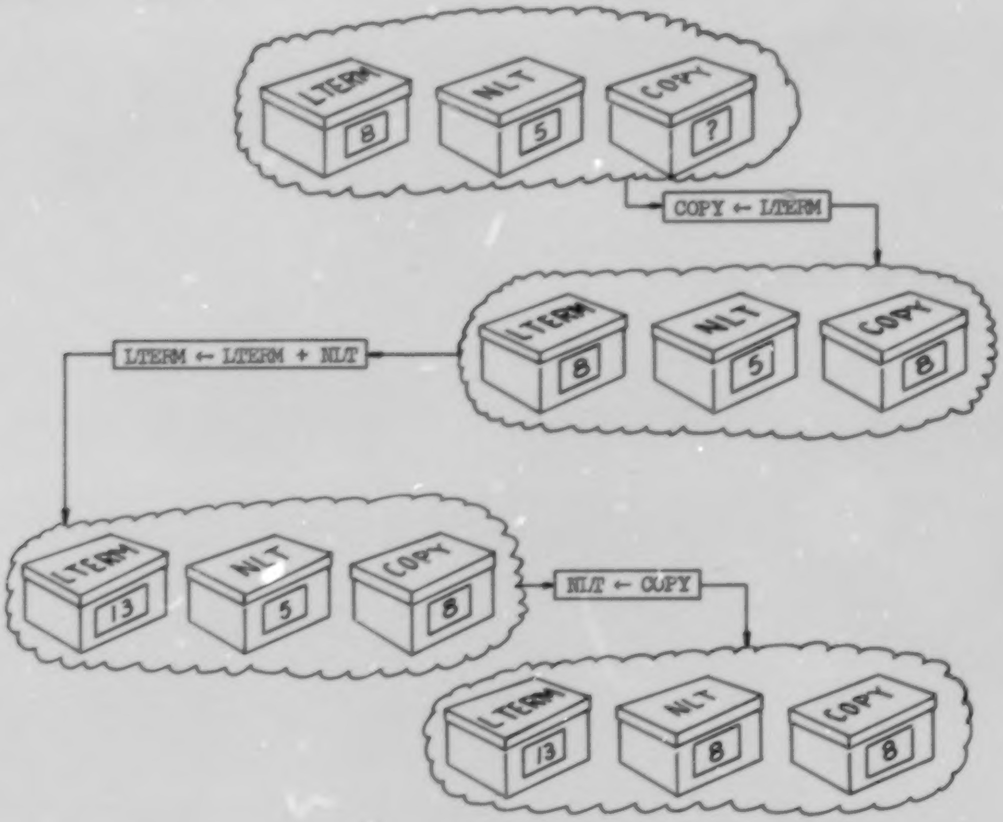
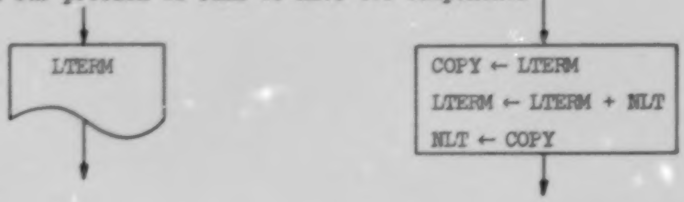
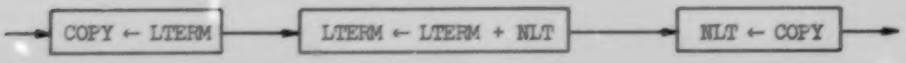

LTERM
NLT
COPY
8
5
?
COPY ← LTERM
LTERM
NLT
COPY
8
5
8
LTERM ← LTERM + NLT
LTERM
NLT
COPY
13
5
8
NLT ← COPY
LTERM
NLT
COPY
13
8
8

Figure 3-11. Successful assignment steps in Fibonacci sequence

Now in our problem we find we have two components


LTERM
COPY ← LTERM
LTERM ← LTERM + NLT
NLT ← COPY

This last assignment box is, of course, shorthand for


COPY ← LTERM
LTERM ← LTERM + NLT
NLT ← COPY

It is not hard now to put our flow chart together. (Figure 3-12)

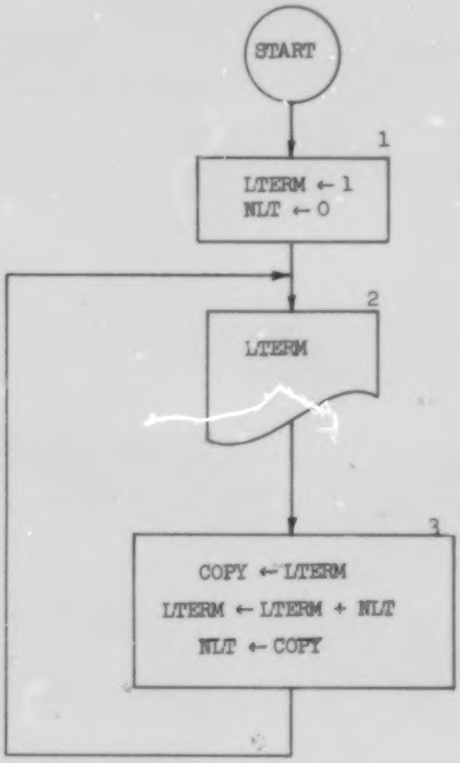

START
1
LTERM ← 1
NLT ← 0
2
LTERM
3
COPY ← LTERM
LTERM ← LTERM + NLT
NLT ← COPY

Figure 3-12. First flow chart for Fibonacci Sequence.

This flow chart has two defects; first, the flow chart shows no way to stop; second, if we wish to know the 657th Fibonacci number we will be forced to count down to the 657th in the output list. Both objections are corrected in the Figure 3-13 flow chart where we introduce an indexing or counting variable, I, and branch to a halt when I assumes the value 1000.

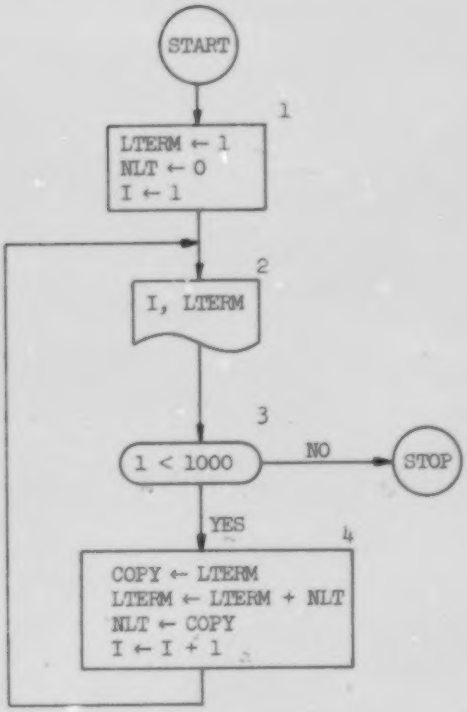
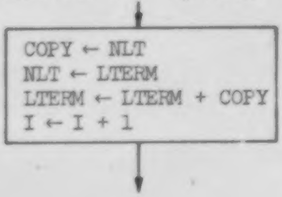

START
1
LTERM ← 1
NLT ← 0
I ← 1
2
I, LTERM
3
1 < 1000
NO
STOP
YES
4
COPY ← LTERM
LTERM ← LTERM + NLT
NLT ← COPY
I ← I + 1

Figure 3-13. Final flow chart for Fibonacci Sequence

We have spent considerable time on this simple example to show how a flow chart can be built up, piece by piece. Moreover, the important idea of a copying variable has been introduced and you have been graphically reminded of the destructive nature of assignment and of the vital importance of order in computer algorithms.

The student should check that the box 4, could be replaced by


COPY ← NLT
NLT ← LTERM
LTERM ← LTERM + COPY
I ← I + 1

Exercises 3-2 Set A

1. In Figure 3-13, why do we choose to assign zero for the initial value of NLT instead of one?

2. Prepare a flow chart to print only the three right-most digits of one hundred terms of the Fibonacci Sequences beginning with the 17th term. Make guesses at how many of these hundred numbers will be even, how many greater than 500, how many between 300 and 400. Save these guesses to see how they compare with results when you run the program on a computer. (Hint: use the Greatest Integer Function.)

In Exercises 3-1, Set B a series of 100 values of T were input from data cards. In the following exercises develop a flow chart for the described operation on this same series of input values of T. Always print some appropriate message identifying the numerical result printed.

3. For each input value after the first value sum and print the two most recently input values of T and their sum. Call this sum TWOSUM.

4. For each input value after the second, sum the most recent value and the value two positions earlier in the series. Print the sums.

5. In each input value after the k^{th} (where the value of k is itself supplied as data and where $3 \leq k \leq 100$) print out the average of either the most recent three values or if the most recent value is lower than its predecessor, print the average of the preceding two values (omitting the most recent one from this average).

6. Prepare a flow chart to calculate and print the first 15 rows of a table according to the following rules:
   1. The table is to have four columns called N, A, B, C.
   2. The values in the first row of the table are 0, 1, 1, 1.
   3. The value of N is one greater than its value in the preceding row.
   4. The value of A is one greater than its value in the preceding row.
   5. The value of B is one greater than the sum of the values of A to and including the preceding row.
   6. The value of C is one greater than the sum of the values of B to and including the preceding row.

This table is of considerable mathematical interest because A is the number of line segments into which a line is divided by N points; B is the number of regions into which a plane is divided by N lines; C is the number of regions into which space is divided by N planes.

The Euclidean Algorithm is a process for finding the greatest common divisor of two integers.

This algorithm is of fundamental importance in mathematics and will be used frequently throughout the book from this point on.

An integer C is a common divisor of integers A and B if it is a divisor of both A and B, i.e., if for some integers m and n

$$A = m \cdot C \quad \text{and} \quad B = n \cdot C \, .$$

The greatest common divisor of A and B is the greatest of all their common divisors.

When we do a long division problem, say dividing 32417 by 1309, our work looks like this:

$$\begin{array}{r|r} & 24 \\ \hline 1309 & 32417 \\ & \underline{2618} \\ & 6237 \\ & \underline{5236} \\ & 1001 \end{array}$$

We recall these names for the numbers appearing here

dividend	(B)	32417
divisor	(A)	1309
quotient	(q)	24
remainder	(r)	1001

The quotient and the remainder are completely determined by the dividend and divisor. In fact, in terms of the greatest integer function

$$q = \left[\frac{B}{A}\right] \quad \text{and} \quad r = B - q \cdot A \, .$$

This last formula shows us that given whole numbers A and B, there are whole numbers q and r so that

$$B = q \cdot A + r \, , \tag{1}$$

And if the condition $r < A$ is imposed, then q and r are uniquely determined.

Now if A, B, q and r are numbers satisfying (1), we will show that the common divisors of B and A are the same as those of A and r. For if C is a common divisor of A and R (i.e., $A = mC$ and $r = nC$ with m and n integers) then

$$B = q \cdot A + r = qmC + nC = (qm + n)C .$$

Of course, $qm + n$ is a whole number so that C is also a divisor of B. If D is a divisor of B and A, i.e., $B = sD$ and $A = tD$, then

$$r = B - qA = sD - qtD = (s - qt)D$$

so that D is also a divisor of r.

In the last paragraph it was shown that any common divisor of A and r is also a divisor of B, and is then a common divisor of B and A; conversely, any common divisor of B and A is also a divisor of r, and is thus a common divisor of A and r. Hence, as we have set out to show, the common divisors of B and A are the common divisors of A and r.

We see then that in the problem of finding the common divisors of B and A we may replace B by r without altering the common divisors. What is gained by this? Simply that we have swapped the original problem for one in which the numbers are smaller, but the answer is the same. We suspect that this swapping process can be repeated, but when does it come to an end? Let us carry out the process completely on the preceding example, but without showing the long divisions. Notice in the following example the pair of values for B and A on the second and succeeding lines are the values of A and r from the immediately preceding line.

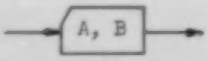
$$\begin{array}{ccccccc} B & = & q & \cdot & A & + & r \\ 32417 & = & 24 & \cdot & 1309 & + & 1001 \\ 1309 & = & 1 & \cdot & 1001 & + & 308 \\ 1001 & = & 3 & \cdot & 308 & + & 77 \\ 308 & = & 4 & \cdot & 77 & + & 0 \end{array}$$

And now the process must terminate because another go-around would call for a division by zero. Each of the following pairs of numbers has just the same common divisors as the preceding pair:

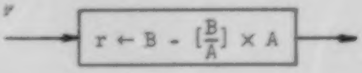
32417 and 1309
1309 and 1001
1001 and 308
308 and 77
77 and 0

The common divisors of 77 and 0 are just the divisors of 77 (since all numbers divide zero) and the greatest of these common divisors is 77 itself. Thus, 77 is the greatest common divisor of 32417 and 1309. The common divisors of these two numbers are just the divisors of 77.

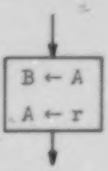
Flow-charting this algorithm is now quite simple. Given values for A and B



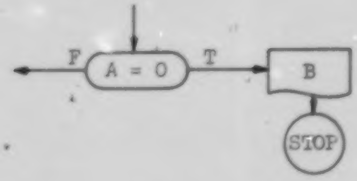
we compute the value of r by



We then replace the values of B and A by the values of A and r, respectively,



and prepare to repeat the process. Except that, if A = 0, we print out the value of B and terminate the process.



These are all the components of the flow chart except for a preliminary check that $A \leq B$ and for a labeling of the result. We now exhibit the assembled flow chart. Note that r plays the role of a copying variable when we need to interchange A and B.

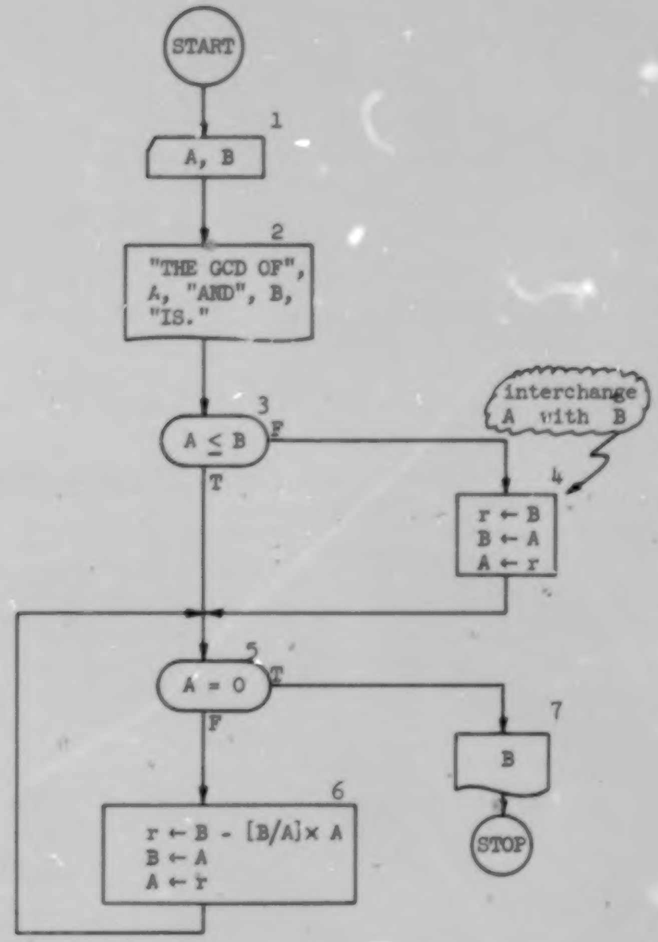
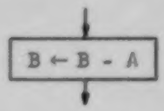
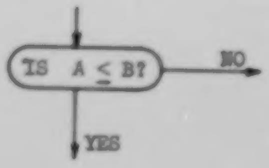

START
1
A, B
2
"THE GCD OF", A, "AND", B, "IS."
3
A ≤ B
F
T
interchange A with B
4
r ← B
B ← A
A ← r
5
A = 0
T
F
6
r ← B - [B/A] × A
B ← A
A ← r
7
B
STOP

Figure 3-14. The Euclidean Algorithm

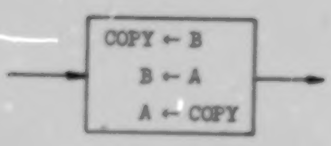
There is another way of flow-charting the Euclidean Algorithm which is less efficient than the above for actual computing, but which has a special charm all of its own. This method depends on the fact that division as we carry it out is repeated subtraction. The remainder r in $B = q \times A + r$ we have seen to be $B - [B/A] \times A$. But it can also be obtained by subtracting A from B enough times (i.e., $B - A - A - A - A - A - A - A - A - A$). This repeated subtraction will be indicated on the flow chart by repeatedly passing through the assignment box:



Before each execution of this box we must check whether A is still less than B.



If not, then we interchange or flip the values of A and B and continue as before.



Now we have only to put the components together, with one eye on the previous flow chart for this problem. For added interest in Figure 3-15, we have included a little fancy printing for you to study.

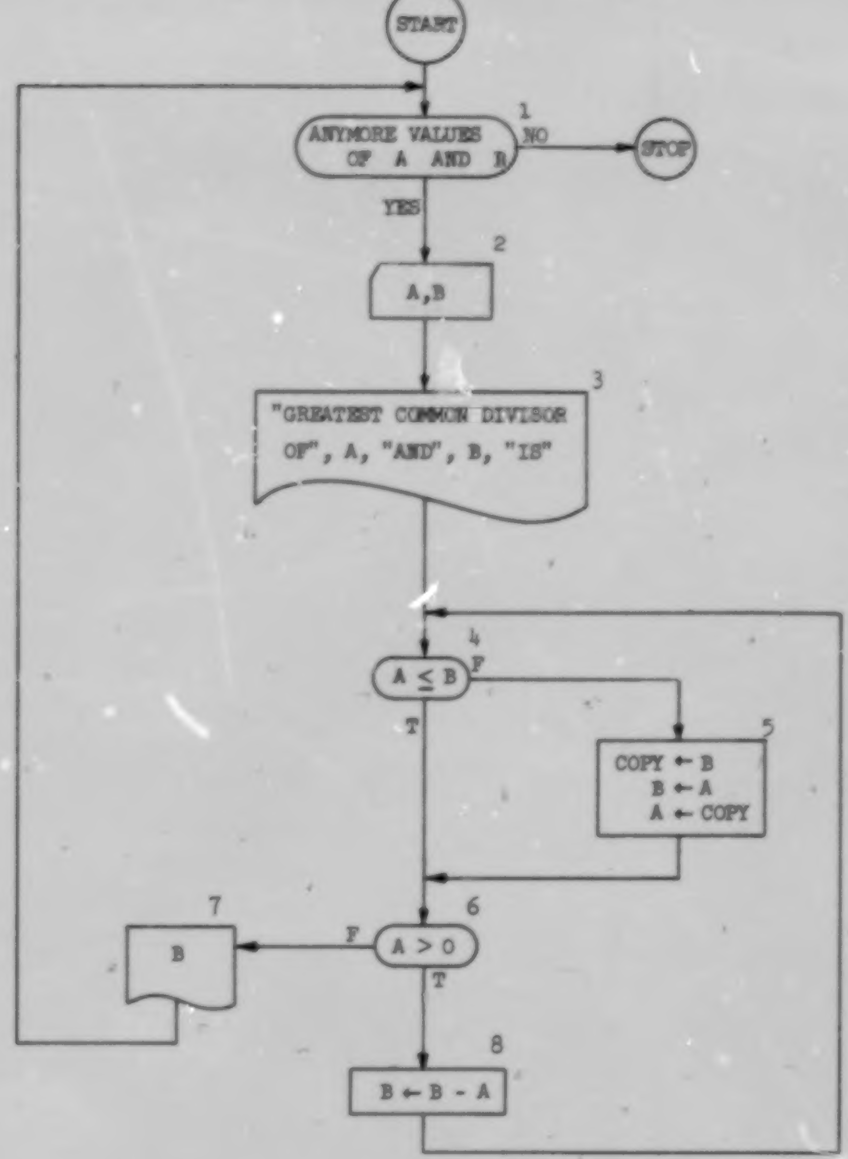

START
1
ANYMORE VALUES OF A AND B
NO
STOP
YES
2
A,B
3
"GREATEST COMMON DIVISOR OF", A, "AND", B, "IS"
4
A ≤ B
F
T
5
COPY ← B
B ← A
A ← COPY
6
A > 0
F
T
7
B
8
B ← B - A

Figure 3-15. Flow chart for Euclidean Algorithm without division (less efficient than Figure 3-14)

Numerical example:

Find the g.c.d. of 16 and 56 using the algorithm of Figure 3-15. $A = 16$ and $B = 56$.

$$\begin{array}{r} 56 \\ -\ 16 \\ \hline 40 \\ -\ 16 \\ \hline 24 \\ -\ 16 \\ \hline 8 \end{array}$$

Since $B < A$, we interchange the values (making $A = 8$, $B = 16$) and continue:

$$\begin{array}{r} 16 \\ -\ 8 \\ \hline 8 \\ -\ 8 \\ \hline 0 \end{array}$$

This time we flip the value of A which is 8, with B which is 0. Note that now $A = 0$, so 8, the value of B, is the g.c.d.

Tracing

The Figure 3-15 flow chart is relatively easy to follow once the student knows what process it is supposed to represent. The reverse problem is often more difficult; that is, one is given a fairly complicated flow chart and is asked to figure out what it is or what it does.

Certain techniques for analyzing a flow chart are therefore frequently indispensable. One called "tracing" is illustrated in Table 3-1. It shows one way we might record vital information about Figure 3-15 as we "trace" our way through (or "execute") the algorithm. This is done here for a particular set of input data, e.g., $A = 16$ and $B = 56$.

It is a good idea to work through this table line by line, observing how we have chosen to record key events as they occur.

There are many ways to produce and show a trace. When searching for the flaws in a really complicated algorithm, either in the flow chart or in an equivalent computer program, professional programmers often trace the algorithm or at least the section under question. Once the algorithm has been connected to a computer program, there are generally easy ways to do this. We can make the computer assist us in tracing by printing out certain vital information, at selected places, while it is executing the algorithm under test.

Table 3-1

Trace of the g.c.d. algorithm in Figure 3-15
for the example where A = 16 and B = 56

After execution of indicated box	Value of B	Value of A	A < B	A > 0	g.c.d. result printed
2	56	16			
4	"	"	true		
6	"	"		true	
8	40	"			
4	"	"	true		
6	"	"		true	
8	24	"			
4	"	"	true		
6	"	"		true	
8	8	"			
4	"	"	false		
5	16	8			
6	"	"		true	
8	8	"			
4	"	"	true		
6	"	"		true	
8	0	"			
4	"	"	false		
5	8	0			
6	"	"		false	
7	"	"			8

Exercises 3-2 Set B

1. Box 4 of Figure 3-14 contains three assignments. What change can be made so that two of these assignments can be eliminated?

2. Draw a flow chart which inputs two non-negative integers C and D and outputs their least common multiple, LCM.
   [Hint: LCM(C,D) = C × D/GCD(C,D)]

Exercises 3-2 Set C

In these eight exercises (x1,y1) and (x2,y2) are to be regarded as the given coordinates of two distinct points P and Q, respectively, neither of which is the origin. Each exercise involves either a straight line passing through the points P and Q or a straight line segment whose endpoints are P and Q. It will help you to know a useful formula for a non-vertical straight line passing through two known (distinct) points P and Q. It is:

$$y - y1 = \underbrace{\left(\frac{y2 - y1}{x2 - x1}\right)}_{\text{the slope}} \times (x - x1)$$

Values of x1, y1, x2, and y2 are to be read as input in that order and then printed back out in the same order. Then the task given in the exercise is to be performed. Draw a complete flow chart for each exercise, including all input and output. Whenever the value(s) requested as output do not exist or fail to be unique, print an appropriate message. In all cases, your flow chart should show a loop to read in more input data.

1. Compute the length of the line segment PQ and output that length preceded by "the length of PQ is".
2. Determine whether the slope of the line PQ is finite. If it is, output that slope preceded by the message "the slope of PQ is". If PQ is parallel to the y-axis, print out a message to that effect.
3. Read as input and print a value for a variable delx. Compute and output the value of dely (if any) for which the point (x1 + delx, y1 + dely) lies on the line (not the line segment) PQ. (See Figure 3-16.)

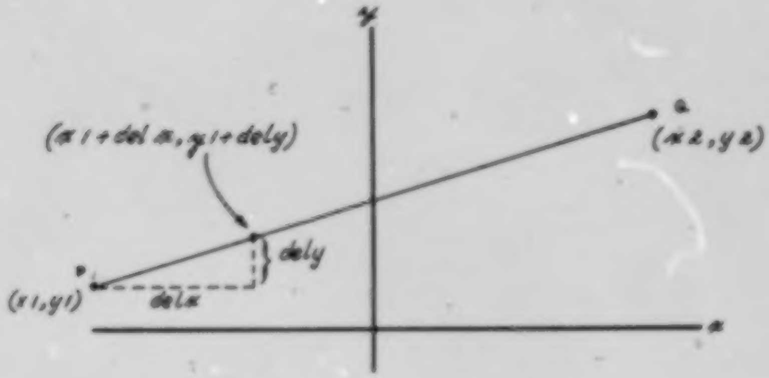


Figure 3-16. Collinearity of three points

4. Read as input and print a number dely. Compute and output the value of delx (if any) such that (xl + delx, yl + dely) lies on the line PQ.

5. Read as input and print a number x. Compute and output the value of y (if any) such that (x,y) lies on the line PQ.

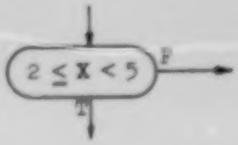
6. Read and print a number y. Compute and output the value of x (if any) such that (x,y) lies on the line PQ.

7. Compute the x-intercept and the y-intercept of the line (not the line segment) PQ. Output both numbers (if any).

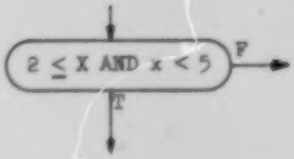
8. Compute and print the x- and y-intercepts of the line segment PQ. Print appropriate messages if the line segment PQ does not intersect the x or the y-axis.

3-3 Compound Conditions and Multiple Branching

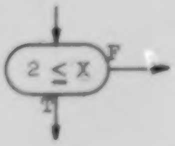
Often one may encounter or wish to write condition boxes such as:

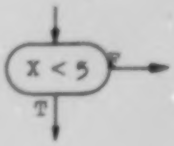


The statement appearing in this box is called a "compound" statement and is obviously equivalent to



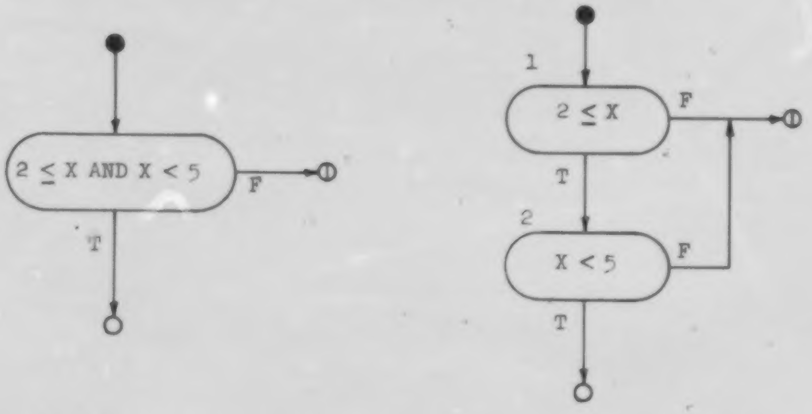
This means that we leave by the botton if both the conditions $2 \leq x$ and $x < 5$ hold. Otherwise, we leave by the side. It is important tó see how to express this compound condition in terms of the simpler components:





In this way we will be able to make flow charts more readily translatable into computer language, the reason being that each condition may have to be tested in a separate step.

Since the compound statement is true only if both simple relations are true and is false if either simple relation is false we can clearly connect the simple condition boxes as in Figure 3-17(b).



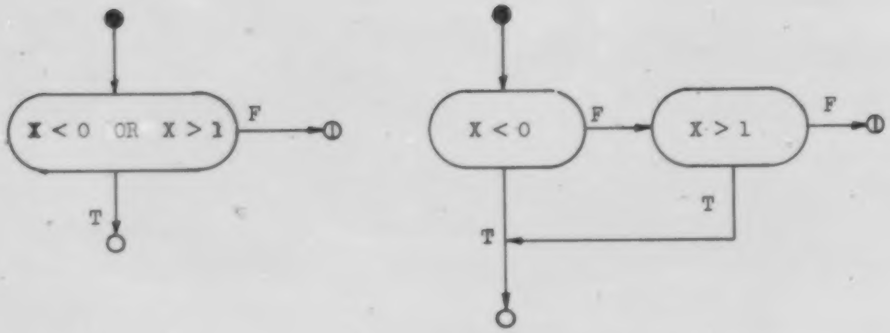
(a) Compound

(b) combination of simple

Figure 3-17. Compound condition box and an equivalent combination of simple boxes

In any flow chart in which it appears, the box in Figure 3-17(a) may be replaced by the combination in Figure 3-17(b), the connections being made as indicated by the arrows. Neither (a) nor (b) is the "more correct." The combination in (b) is the more detailed and hence the more readily translated into machine language. In that respect (b) is better. But on the other hand, the single box in (a) is more easily scanned by a reader who wishes to know what the flow chart is doing.

In constrast to this example where we want to know whether both of two conditions are true, there are places where we might want to know whether either of two conditions is true. The decomposition of the latter type of compound condition into simple conditions is shown in Figure 3-18(b).



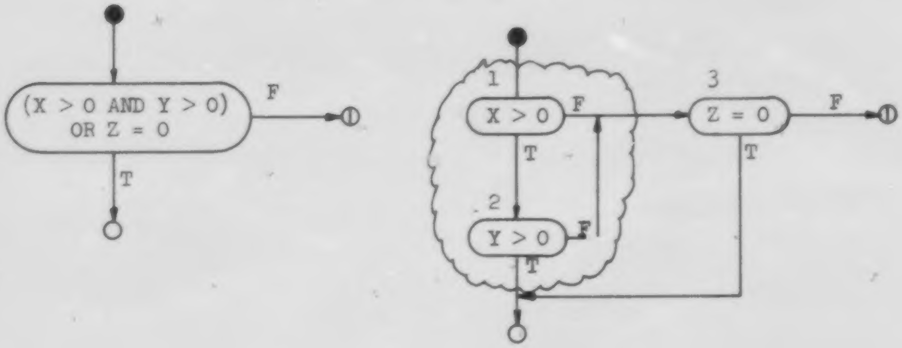
(a) Compound

(b) Combination of simple

Figure 3-18. Another compound condition box and its equivalent combination of simple boxes

Clearly, compound condition boxes could grow to any degree of complexity demanded by the problem, with any number of conditions to be satisfied and any number of variables involved.

For example, if we want to know when both X and Y are positive or Z is zero we can draw the compound box and its decomposition as in Figure 3-19.



(a) Compound

(b) Combination of simple

Figure 3-19. Composition of condition boxes

Notice that the decomposition in Figure 3-19 can be accomplished in two stages. We first use the method shown in Figure 3-18 to decompose the "or" statement in 3-19(a) to obtain

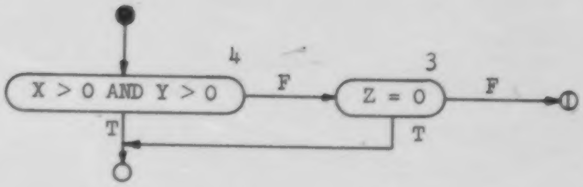
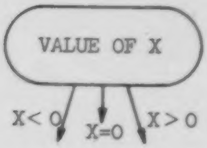


Figure 3-19(c)

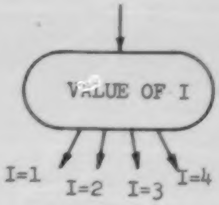
Now the method of Figure 3-17 is used to replace Box 4 of Figure 3-19(c) by the "cloud" in Figure 3-19(b).

A compound condition box may be regarded as shorthand for a combination of simple condition boxes. There is another type of shorthand associated with condition boxes which can be extremely helpful in the process of gradually building up complicated flow charts. This shorthand technique is designated by the name of "multiple branching."

To indicate multiple branching we will draw compound condition boxes with several exits. Each exit must be clearly labeled to show what condition would cause its use. For example:



and



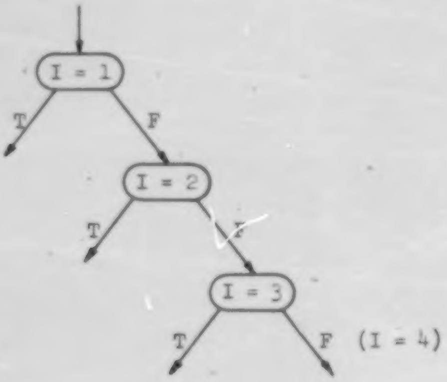
Two important warnings are in order:

(1) The conditions on the exits must not be overlapping. If one exit were labeled "$3 < X < 7$" and another were labeled "$5 \leq X < 10$", then if we come into this box with a value of X between 5 and 7, we will not know which branch to take on leaving.

(2) All possibilities must be exhausted. If the conditions on the exits were "$X \leq 3$" and "$6 \leq X < 9$" and "$9 \leq X$", then if we come into the box with a value of X between 3 and 6 we will have no way to get out. Then we will really be in a box!

An example of the usefulness of multiple branching is provided by the example in Section 3-1 of tallying test grades as flow-charted in Figure 3-7. The way in which this same problem might have been handled with multiple branching is shown in Figure 3-20. We simply "collapse" the chain of two 2-way branches, (Boxes 4 and 5 of Figure 3-7) into a single 3-way branch (Box 4 of Figure 3-20).

We should note in passing that any box indicating a multiple branch of n ways can be broken down into a chain of n - 1 2-way branches. Thus the 4-way branch on the value of I may be viewed in more detail as the chain of three 2-way branches:



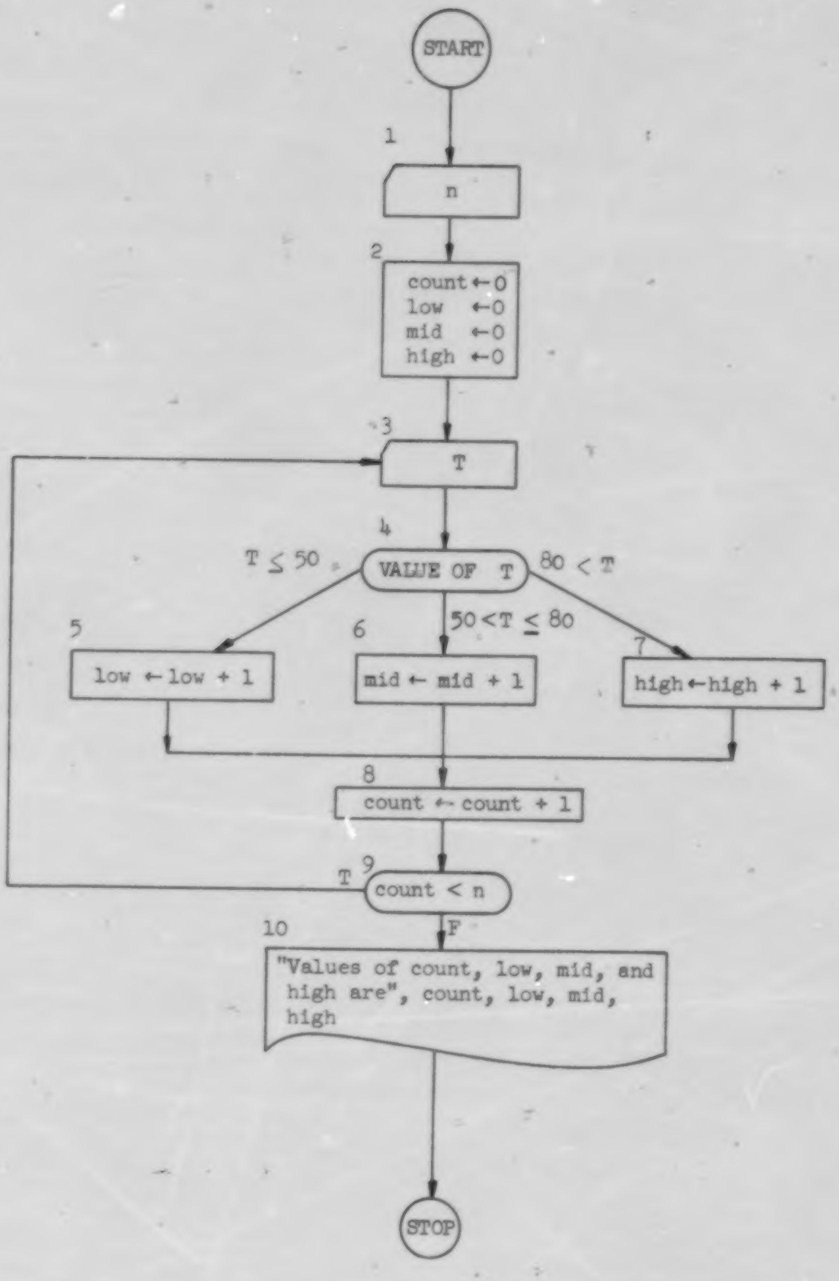

START
1
n
2
count ← 0
low ← 0
mid ← 0
high ← 0
3
T
4
VALUE OF T
T ≤ 50
50 < T ≤ 80
80 < T
5
low ← low + 1
6
mid ← mid + 1
7
high ← high + 1
8
count ← count + 1
9
count < n
T
F
10
"Values of count, low, mid, and high are", count, low, mid, high
STOP

Figure 3-20. Use of a three-way branch

In the normal course of events this multiple branching flow chart would have been given first. It represents our first formulation of the problem. After we had first drawn this flow chart we would then have given our attention to the problem of decomposing the multiple condition box into a combination of simple conditions.

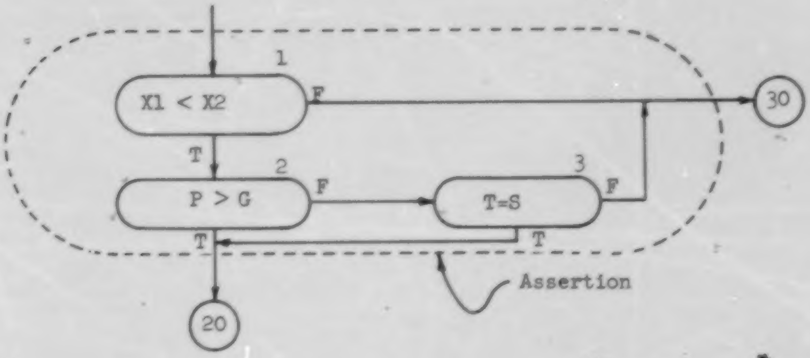
Exercises 3-3

In each of the next seven exercises your job is to construct the flow chart equivalent to the given assertion using only simple condition boxes. The "true" path of the assertion should lead to Box 20 and the "false" path to Box 30.

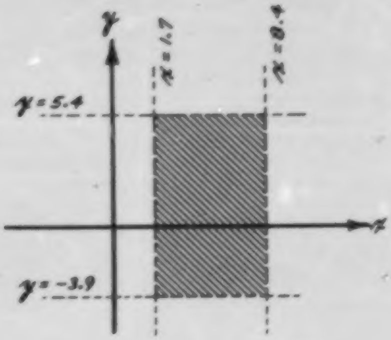
Example:

The assertion is: x1 is less than x2 and either P exceeds G or T equals S or both.

The required flow chart is:



1. x lies between 2 and 7, inclusive.
2. Either 7 is less than Q or 7 is less than R or 7 is less than S.
3. x lies between 1.7 and 8.4, and y lies between -3.9 and +5.4.

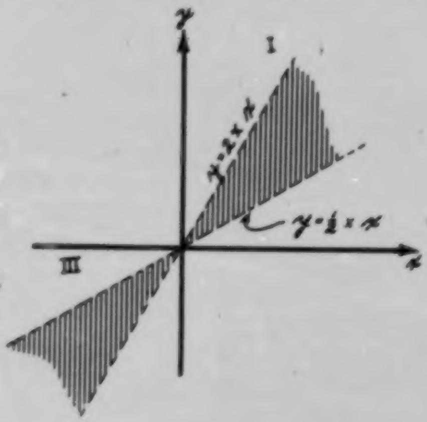


4. Given the shaded region inside the two straight lines whose equations are

$$y = 2 \times x$$

and

$$y = \frac{1}{2} \times x$$

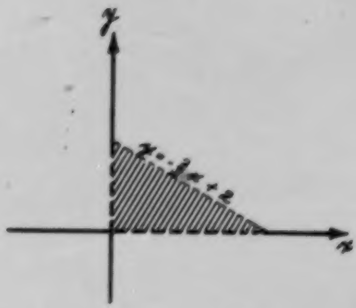


(a) The point (x1,y1) lies inside the shaded region of quadrant I.

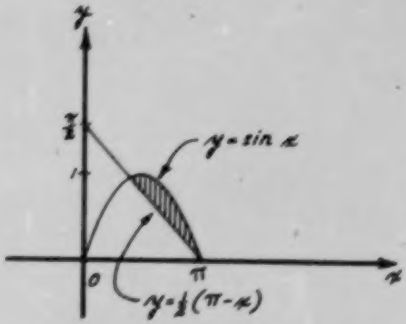
(b) The point (x1,y1) lies inside the shaded region of quadrant III.

(c) The point (x1,y1) lies somewhere inside the shaded region of quadrants I or III.

5. The point (x1,y1) lies inside the shaded triangle in the first quadrant formed by the straight line $y = -\frac{2}{3}x + 2$ and the coordinate axes.



6. The point (x1,y1) lies in the shaded area (or on its boundaries) formed by the curve, y = sin x and the straight line, $y = \frac{1}{2}(\pi - x)$.



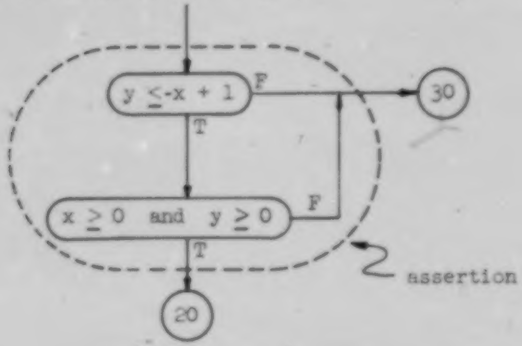
7. The intersecting straight lines

$$y = 4 \times x - 12 \quad \text{and} \quad y = -4 \times x + 16$$

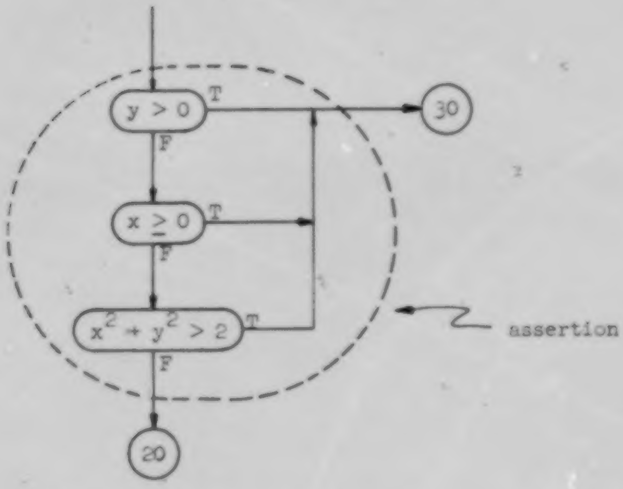
determined four regions, one of which, region A, lies entirely in the upper half-plane. We assert that the point (x1,y1) lies in the interior of region A or on its boundary.

For each of the following flow chart assertions, certain x, y pairs lead to Box 20. These pairs define a region in the x-y plane. Your job is to draw the graph of this region.

8.


$y \leq -x + 1$
F
T
30
$x \geq 0$ and $y \geq 0$
F
T
20
assertion

9.


$y > 0$
T
F
30
$x \geq 0$
T
F
$x^2 + y^2 > 2$
T
F
20
assertion

10. Draw a flow chart which computes and prints the numbers 1, 2, 3, or 4 as a message to indicate in which quadrant a point P lies. The coordinates (xl,yl) of P are given. What happens if P lies on one or both axes?

11. We return to the carnival wheel problem (Exercise 6, Section 2-5, No. 9). We suppose now the rule is modified. Recall p, the number of points won or lost, was originally a straight line function of k, the position number in the repeated group of four sectors. We now want a new point rule where p is an arbitrary function of k. For example,

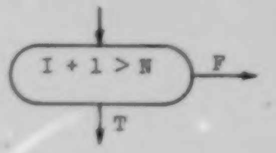
k	Old point rule	New point rule
0	lose 30	lose 20
1	lose 10	lose 30
2	win 10	win 0
3	win 30	win 50

Draw a revised flow chart to show p as a function of the same data pair S and m but with the new point rule given above. (S is the sector position of the wheel at rest, and m is the number of sector positions the wheel is spun.)

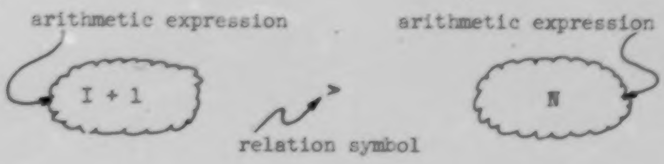
3-4 Precedence Levels for Relations

We pause here to look in a more formal way at the statements which we have been writing inside the oval condition boxes. When the lines emanating from the oval are marked true (or T) and false (or F), perhaps a more appropriate term for the statement which appears inside is an assertion.

Consider the condition box



(More complicated conditions may, as we have seen, be represented through combinations of this type.) The condition inside the above box consists of two arithmetic expressions with a "relation symbol" between them.



The complete list of "relation symbols" used in this text is

$$<, >, \leq, \geq, =, \neq$$

Such a symbol together with a constant on each side amounts to an assertion that a certain relation holds between these numbers. The assertion may be either true or false. For example:

$$7 > 5$$

is certainly true, while

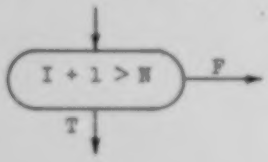
$$8 = 7$$

is obviously false. We may also see such a relation symbol between two arithmetic expressions such as:

$$I + 1 \geq N$$

This states that a certain relation holds, not between the expressions "I + 1" and "N", but between their values.

Suppose we have a flow chart with such a box in it as this:



During the computation indicated by the flow chart we may pass through this box many times. Sometimes the assertion indicated by the "relational expression" in the box will be true and sometimes false. And the truth or falsity determines the exit by which we leave.

Let us look at our method of determining whether the assertion is true or false.

(1) Look up the current values of the variables.

(2) Evaluate the arithmetic expressions on either side of the relation symbol.

(3) Determine whether the relation in question holds between the numerical constants obtained in (2).

It follows, for example, that the expression

$$X^2 + 2 \times X + 1 < 2 \times A \times B$$

will be read as though parentheses were inserted as follows:

$$(X^2 + 2 \times X + 1) < (2 \times A \times B).$$

We can convey the same idea by saying that when reading expressions having no parentheses, relational symbols have a lower precedence than any of the arithmetic operators. We can expand the precedence table, Table 2-4, to include the relational symbols.

Table 3-2

Precedence Levels for Relational Expressions

Levels		Symbol
High	First	"exponentiation" or "raising to a power"
	Second	×, /, - (unary)
	Third	+, - (binary)
Low	Fourth	$<$, $>$, $\leq$, $\geq$, $=$, $\neq$

Nothing need be said about scanning left-to-right for symbols of the fourth level, since in a properly written expression there can be, at most, one such symbol. Such an expression as

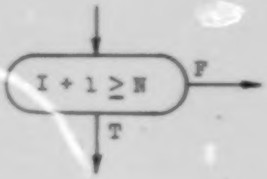
$$3 < X < 5$$

(frequently encountered in mathematics) is actually a compound expression, i.e., in this case

$$3 < X \quad \text{and} \quad X < 5$$

We have seen how to deal with such compound statements in the preceding section.

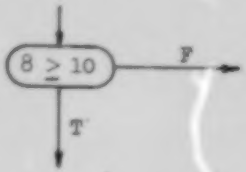
Perhaps you would be interested in seeing how a machine might find out whether one of the above inequalities is true or false. Consider, for example, the SAMOS computer of Appendix A. It has only the one branching instruction BRANCH ON MINUS. We consider the condition box at the beginning of this section,



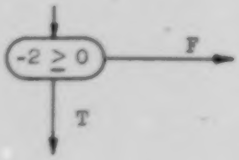
with the current values of the variable, given by

Variable	I	N
Current value	7	10

First the values of the variables in the condition box are looked up and the arithmetic expressions on either side of the relation symbol are evaluated. The condition box may now be visualized as:



Because $8 \geq 10$ is equivalent to the relation $8 - 10 \geq 0$, the expression $8 - 10$ is evaluated and we may now visualize the condition box as:



The machine determines the truth or falsity of the relation $-2 \geq 0$ by examining the first character in the numeral on the left. This character being a minus, we "branch on minus," that is, we go to an address specified in the branching order to pick up our next instructions corresponding to the false side of the flow chart box. Otherwise, the next instruction after the branch on minus will be executed. This corresponds to emerging from the true side of the flow chart box.

3-5 Subscripted Variables

We come now to the second of the powerful tools referred to in the opening paragraph of this chapter--the subscripted variable.

We admit as variables inscriptions of the sort:

$$X_1, X_2, X_3, X_4, X_5 .$$

Here the thing occupying the position of X may be the inscription for any properly written variable while the subscript must be an integer.

Each subscripted variable is provided with a window box as with ordinary variables as suggested in Figure 3-21.

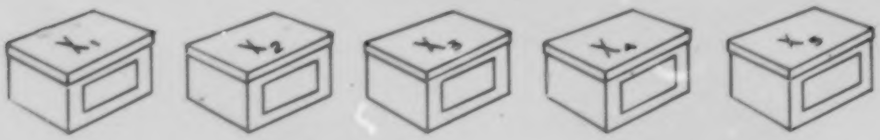


Figure 3-21. Window boxes for subscripted variables

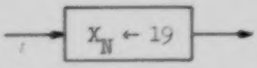
We do not introduce these subscripted variables just for the purpose of having more variables available. If that were all we wanted, we could use:

X1, X2, X3, X4, X5 .

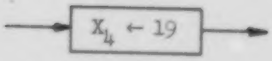
The application of subscripted variables lies in our ability to write expressions like

$$X_N$$

in our flow chart boxes. N is a variable which can have only an agreed set of consecutive values, like 1, 2, 3, 4, and 5. Let us see how we interpret such an inscription. Suppose we find in a flow chart the assignment box



Evidently, we are supposed to put 19 somewhere. But where? If we look at the window boxes of Figure 3-21, we find boxes labeled X_1, X_2, X_3, X_4, and X_5; but none labeled X_N. We do the obvious thing. We look up the current value of N. Say it is 4. Then we interpret the assignment box shown above to mean

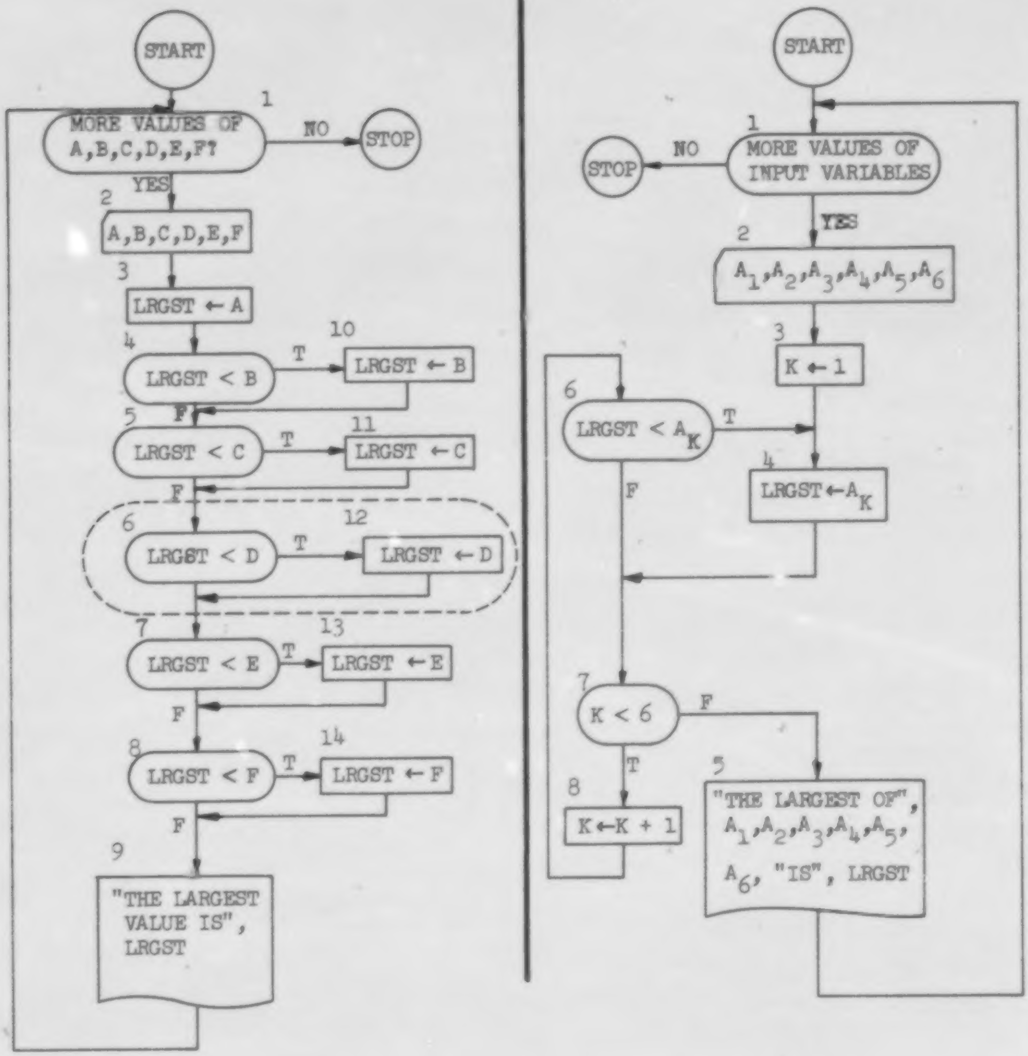


Thus X_N is a subscripted variable which unambiguously designates one window box from the set inscribed with X_1, X_2, X_3, X_4 or X_5. Which window box is designated depends on the current value of the variable N.[†]

The domain of permissable values of the subscript N can, of course, be as large as is necessary. Generally we will limit N to the non-negative integers. The real power of subscripted variables becomes evident when we consider problems having a large number of related variables. The next example begins to illustrate the power of this notation.

Consider the problem of finding and printing the largest value of six input variables. We treated this problem with three input variables in Section 3-1. The flow chart we will generalize is given in Figure 3-5(b). We give the generalized flow chart in Figure 3-22(a) with no further explanation.

[†] Note that X_N is a very different "animal" from XN. The latter case can only designate one window box--that inscribed with XN.


START
1
MORE VALUES OF A,B,C,D,E,F?
NO
STOP
YES
2
A,B,C,D,E,F
3
LRGST ← A
4
LRGST < B
T
10
LRGST ← B
F
5
LRGST < C
T
11
LRGST ← C
F
6
LRGST < D
T
12
LRGST ← D
7
LRGST < E
T
13
LRGST ← E
F
8
LRGST < F
T
14
LRGST ← F
F
9
"THE LARGEST VALUE IS", LRGST
START
1
MORE VALUES OF INPUT VARIABLES
NO
STOP
YES
2
A1,A2,A3,A4,A5,A6
3
K ← 1
6
LRGST < AK
T
4
LRGST ← AK
F
7
K < 6
F
T
8
K ← K + 1
5
"THE LARGEST OF", A1,A2,A3,A4,A5, A6, "IS", LRGST

(a) without

(b) with

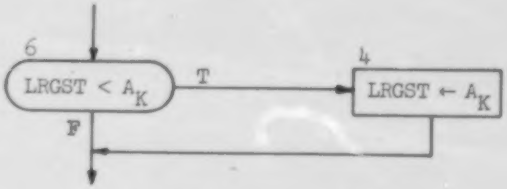
Figure 3-22. Flow chart for finding largest of six numbers without and with subscripted variables

We see a certain monotonous repetition in this flow chart. Think how much worse the situation would have been if there were a hundred input variables instead of only six. The problem would get out of hand (as well as off the paper).

We will now see how to treat the same problem with subscripted variables. We let the input variables be

$$A_1, A_2, A_3, A_4, A_5, A_6 .$$

We have put a dashed line around one of the "blocks" which go into making up this flow chart. The general form of such a block in our subscripted variable notation would be:



With suitable values assigned to K this configuration can represent any of the five "blocks" in Figure 3-22(a). But how do we get to the next step? Clearly, unless K already has the value 6, we augment K by 1 and come back to the top of the block. So far we have:

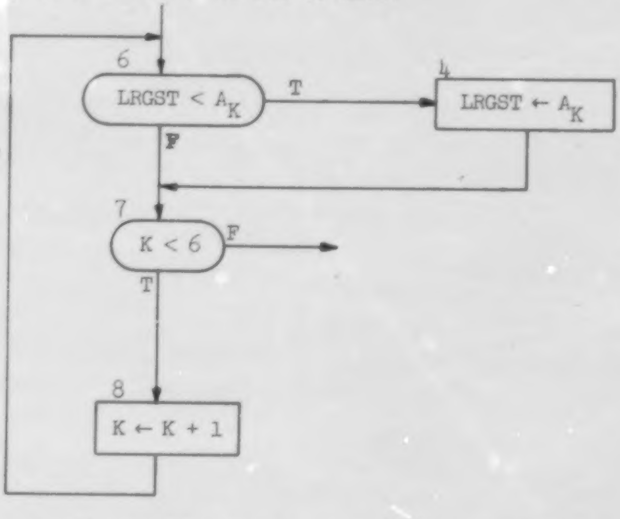


Figure 3-23. Partial flow chart for largest of a set of numbers

What we have so far described represents the body of the flow chart. All that remains is to attach the head and tail.

To start, we must input the data, assign the initial value of 1 to the variable K and hook in at the top of Box 4 of Figure 3-23. If we leave Box 7 at F we print the current value of LRGST and go back for more data (if any). The complete flow chart is given in Figure 3-22(b).

Careful study of the development of the flow chart in Figure 3-22(b) and comparison with that in Figure 3-22(a) will show better than any number of words the importance of subscripted variables and the way we use them.

An important thing to observe about the flow chart of Figure 3-22(b) is that it would only be changed in the most minor way if we had 30 input variables rather than a mere six. In Box 7, the 6 would be replaced by 30. In the input and output boxes, 2 and 5, $A_1, A_2, A_3, A_4, A_5, A_6$ would be replaced by $A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10}, A_{11}, A_{12}, A_{13}, A_{14}, A_{15}, A_{16}, A_{17}, A_{18}, A_{19}, A_{20}, A_{21}, A_{22}, A_{23}, A_{24}, A_{25}, A_{26}, A_{27}, A_{28}, A_{29}, A_{30}$.

We can avoid all of this writing by introducing a more compact notation. The real point is that the structure of the flow chart, i.e., the way the boxes are connected, does not depend on the amount of data.

In the input and output boxes we are really dealing with a set of variables and there is a convenient notation in common use which we can adopt. The notation:

$$\{A_K, \quad K = 1(1)30\}$$

is a shorthand equivalent to listing every element A_1 through A_{30}. Explanation of this notation is given below:

$$\{A_K, \quad K = 1(1)30\}$$

initial value of subscript.

amount by which subscript is incremented (i.e., size of steps) in advancing from element to element in the list.

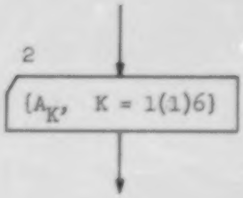
cut-off point. No subscripts larger than this allowed.

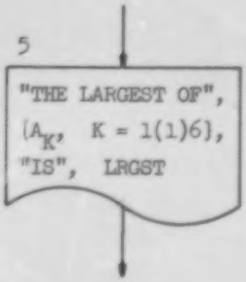
Example: The notation $\{A_J, \quad J = 7(5)23\}$

denotes $A_7, A_{12}, A_{17}, A_{22}$.

This notation may be used either in an input or in an output box. In most uses in this text, the initial value and the increment will both be 1.

The input and output boxes, 2 and 5 of Figure 3-22(b) could be replaced by:



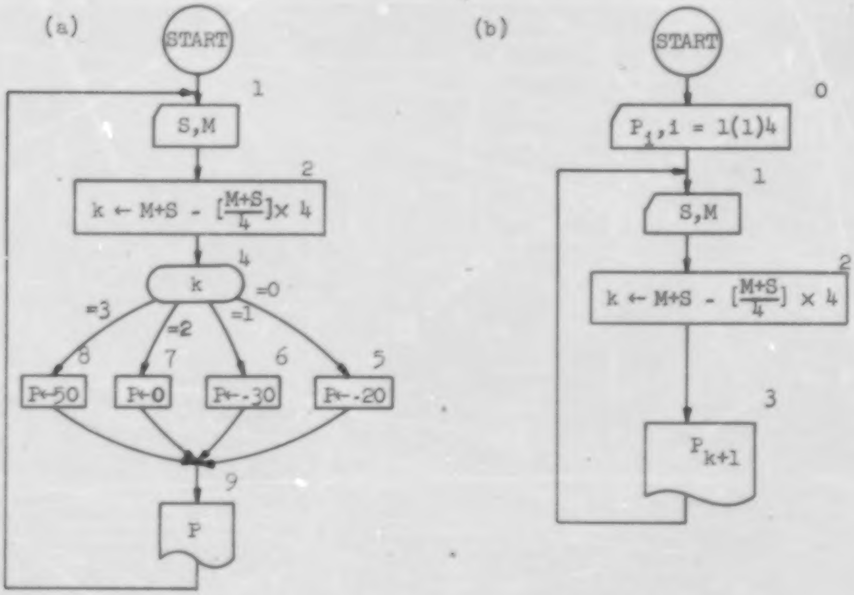

5
"THE LARGEST OF",
{A_K, K = 1(1)6},
"IS", LRGST

Exercises 3-5 Set A

Problems 1 and 2: When we revised the carnival wheel problem in Exercise 11, Section 3-3, we employed a multi-way condition box to model the new point rule. In Figure 3-24(a) you see one way to achieve that objective.

A student now proposes an alternative solution shown in Figure 3-24(b). He claims it is simpler, equivalent, and an inherently more general solution. Study these two flow charts carefully and answer the two questions given below.

1. Under what circumstances are the flow charts equivalent?
2. In what sense can (b) be construed to be more general than (a)?


(a)
START
1
S,M
2
k ← M+S - [(M+S)/4] × 4
4
k
=0
=1
=2
=3
8
7
6
5
P←50
P←0
P←-30
P←-20
9
P
(b)
START
0
P_i, i = 1(1)4
1
S,M
2
k ← M+S - [(M+S)/4] × 4
3
P_{k+1}

(a) Carnival wheel with a conditional

(b) Carnival wheel with subscripts

Figure 3-24

Problems 3 through 7: The flow chart in Figure 3-25 is an algorithm which accomplishes the following steps:

(a) Inputs a number c

(b) Inputs 100 numbers $b_1, b_2, \ldots, b_{100}$

(c) Determines and prints a list of the b_i which satisfy the relation

$$b_i \geq c$$

Study the flow chart carefully and answer the following questions.

3. How many times is Box 6 executed?

4. How many times is Box 8 executed?

5. Under what circumstance will Box 10 be executed? The remark is made that ANY is a "switch variable"; that is, it is used like railroaders use a rail switch. Explain.

6. Is it really necessary for there to be more than one value of b in memory at any given time in order to achieve the same output objectives for this program? Another way of asking this question is, "Are subscripts really necessary in this algorithm?" If your answer is no, redraw the flow chart accordingly putting a check mark next to each box you change.

7. How would you modify either Figure 3-25 or your modified version, resulting from 6, to generalize the flow chart so that instead of reading 100 elements for b we read any given number n of them?

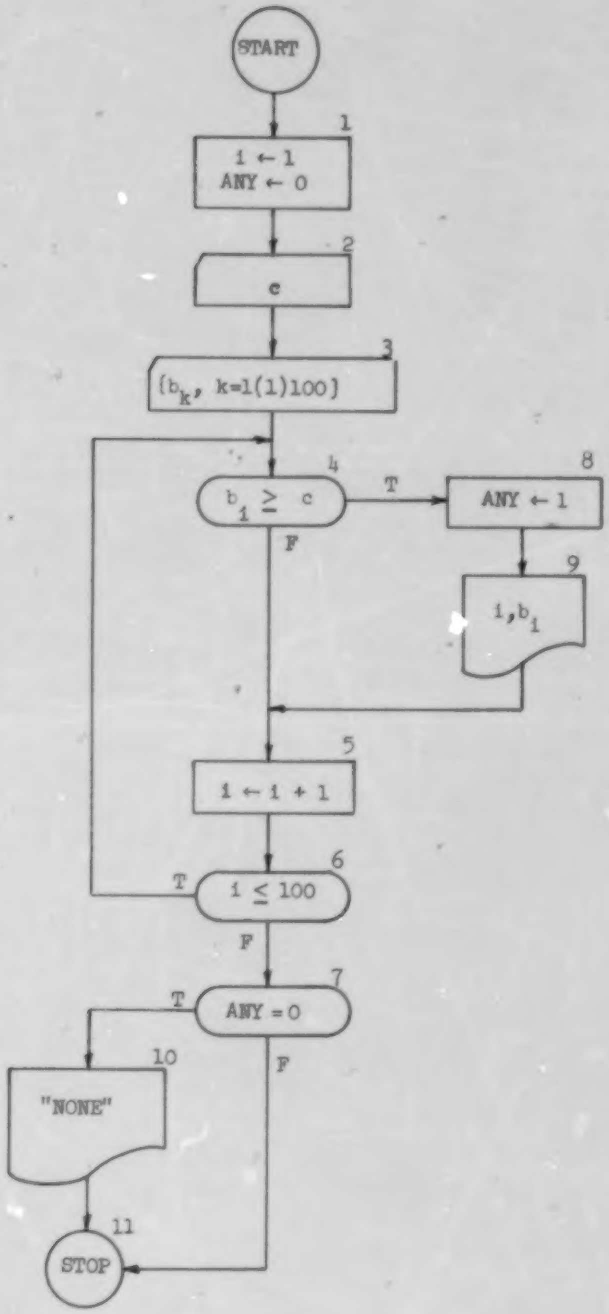

START
1
i ← 1
ANY ← 0
2
c
3
{b_k, k=1(1)100}
4
b_i ≥ c
T
F
8
ANY ← 1
9
i,b_i
5
i ← i + 1
6
i ≤ 100
T
F
7
ANY = 0
T
F
10
"NONE"
11
STOP

Figure 3-25. An algorithm to study

8. Draw a flow chart for inputting $\underline{n}$ and a vector $a_0, a_1, \dots, a_n$. The a's are considered to be the coefficients of the polynomial

$$a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

and $\underline{n}$ is its apparent degree. However some or all of the coefficients may be zero. Construct a flow chart to determine the actual degree, m, of the polynomial. Of course $m \leq n$ and m can be determined by searching the set of coefficients for the non-zero element with the highest subscript. Output m and the coefficients from a_0 through a_m inclusive. If all the coefficients are zero, don't print any coefficients but let the printed value of m be -1.

Additional remarks on subscripts

As we have said, we view X_N as designating one of a set of window boxes inscribed with $X_1, X_2, X_3, \dots$. Which box is designated depends on the current value of the variable N. Now, what do we mean by X_I? By the same reasoning, X_I designates one of the boxes $X_1, X_2, X_3, \dots$ depending on the current value of the variable I. X_N and X_I may designate different boxes (if $N \neq I$) or may designate the same box (if $N = I$).

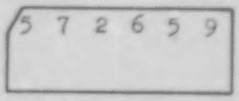
Now, what do you think X_{N+1} should mean? Apparently it should designate one of the boxes inscribed with $X_1, X_2, X_3, \dots$. Which of these is designated should depend on the current value of the variable N. Suppose the value of N is 3. Then $N + 1$ is 4 and X_{N+1} really means X_4. From this example you will correctly guess that arithmetic expressions can be used as subscripts. However, procedural languages sometimes place limits on the complexity of expressions used as subscripts (see your language supplement). In this text we will normally avoid expressions more complicated than $N + 1$ (or $N + K$) as subscripts.

In summary, if the subscript of a variable is an expression, it must be possible to compute the value of this subscript each time the subscripted variable is encountered in a flow chart box. The subscript expression must be integer-valued. Like any expression, a subscript expression is "computable" if we have previously assigned values for every variable that appears in the subscript expression.

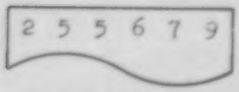
Sorting Example

Frequently in computing we have to put numbers (or other things, like names) into some kind of order. This, "sorting", seems like a very simple thing but the problem arises so often as part of larger problems that much effort has been spent to be able to do sorting as efficiently as possible. Many algorithms have been invented and many refinements made for this purpose. Now we will develop one of many possible algorithms for sorting. We will study other sorting algorithms later.

In sorting the problem is this: If we input a set of numbers:



we should output:



Consider a list of input variables with values:

A_1	A_2	A_3	A_4	A_5	A_6
5	7	2	6	5	9

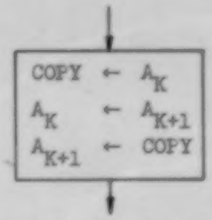
Scan the values from left to right until we encounter the first place where the values decrease. (If there is no such decrease, then the values are already in increasing order.) In the above example, we find this first decrease when going from A_2 to A_3. Interchange these values:

A_1	A_2	A_3	A_4	A_5	A_6
5	2	7	6	5	9

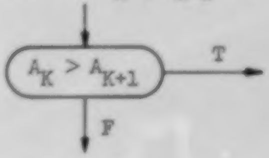
What next? Well, we seem to have done some good. So, let's treat this list just like a brand new one. That is, go back to the beginning and scan from left to right, etc.

This almost seems too simple to work! Nevertheless, we observe that as long as the list is not in increasing order, there will always be another interchange to do. Each interchange affects the relative order of just one pair of values and since there are only finitely many such pairs, the algorithm must terminate. Perhaps you'd like to try the process with some playing cards.

Next, to put this algorithm on a flow chart. The basic idea is the interchange of A_K and A_{K+1} which, experienced as we have become, we know to represent as


COPY ← A_K
A_K ← A_{K+1}
A_{K+1} ← COPY

We execute this interchange only if $A_K > A_{K+1}$. Thus, the condition box:



If false, we go to the next position in the list

and repeat the test (i.e., return to the condition box). On emerging from the interchange box, we set K back to 1 and start over. We now have the skeleton of Figure 3-26.

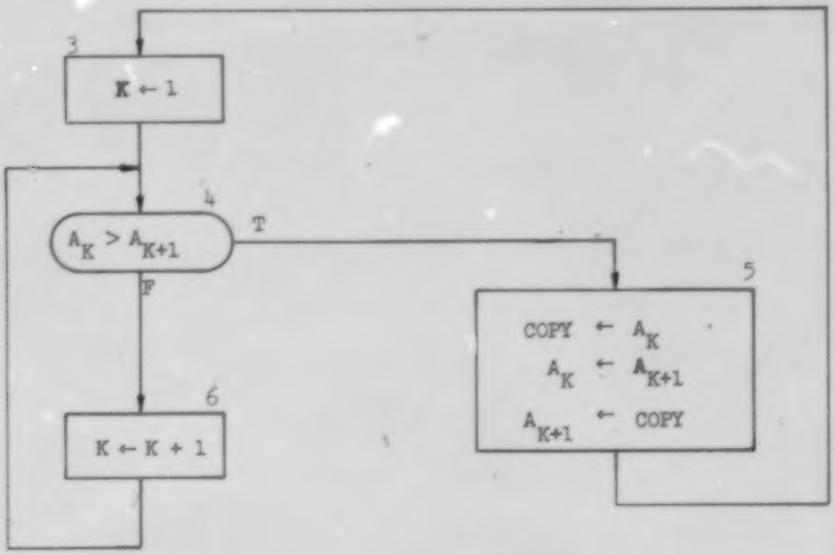

3
K ← 1
4
$A_K > A_{K+1}$
T
F
5
COPY ← A_K
A_K ← A_{K+1}
A_{K+1} ← COPY
6
K ← K + 1

Figure 3-26. Skeleton of a sort

Only input, output and stopping mechanism are needed. We should also decide on how large a list of numbers the flow chart should be set up to handle. One time we may want to sort 13 numbers, another time 200, or perhaps 1000. Why not let the variable N denote the length of the list? This is all put together in Figure 3-27.

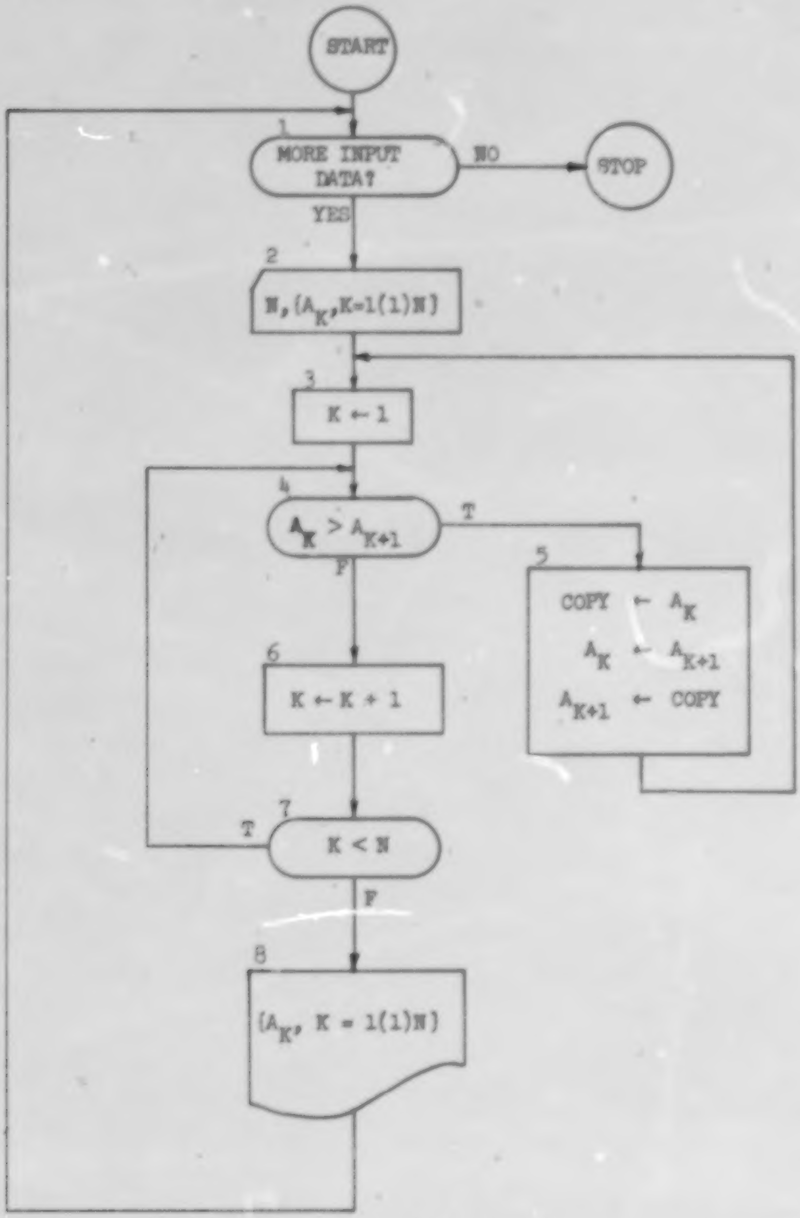

START
1
MORE INPUT DATA?
NO
STOP
YES
2
N, (A_K, K=1(1)N)
3
K ← 1
4
A_K > A_K+1
T
F
5
COPY ← A_K
A_K ← A_K+1
A_K+1 ← COPY
6
K ← K + 1
7
T
K < N
F
8
(A_K, K = 1(1)N)

Figure 3-27. Sort

Before we leave this section there is a terminology we should like to introduce in connection with subscripted variables. Let us suppose we have a subscripted variable such as

$$(X_I, \quad I = 1(1)6)$$

It is then customary to refer to the list (or linear array):

$$X_1, X_2, X_3, X_4, X_5, X_6$$

or the list of values of these variables:

$$7, \quad 9.2, \quad 32, \quad 17, \quad -2.73, \quad 0$$

as a "vector." The individual entries in this list are referred to as the "components" of the vector. This is in agreement with mathematical usage. Engineers and physicists often speak of a vector as having a magnitude and direction but that view is really just a special example of our mathematical description of a vector.

Mathematical notation requires that our vectors be enclosed in parentheses as

$$(X_1, X_2, X_3, X_4, X_5, X_6) \ .$$

We will not insist on these outer parentheses in our computer work.

We will frequently use such terminology as, "The vector X," to designate the list,

$$X_1, X_2, X_3, X_4, X_5, X_6 \ .$$

Exercises 3-5 Set B

This group of exercises concerns the sorting algorithm given in Figure 3-27.

1. Suppose you wanted to test the algorithm by determining if the list

   7 2 -5 4

   will be properly sorted in ascending order, i.e.,

   -5 2 4 7

   (a) What are the values of the input data at Box 2?

(b) With these input data trace through the algorithm beginning at box 3, showing the box numbers in the sequence they are actually executed until box 8 is reached. Use a table like the one given here. It is partially filled for this problem to help you get started.

box / sequence	3	4	5	6	7	8	assigned value of K
1	✓						1
2		✓					
3			✓				
4	✓						1
5		✓					
6				✓			2
7							
8							
9							
.							
.							
.							

Scratch pad for a vector

1	~~7~~ 2
2	~~2~~ 7
3	-5
4	4

(c) How many times in this sequence has a flow chart box (including box 8) been executed before returning to box 1?

(d) How many times is box 4 reached?

2. By now you should be thoroughly convinced this algorithm will work every time. Suppose the values to be sorted are

-9 5 9 12

That is, they are already in ascending order. How many times will box 4 be executed before box 8 is reached?

3. What if the input values are already sorted but in opposite order, say

12 9 5 -9

How many executions of box 4 ?

Save your results for problems 1, 2, and 3. In the next chapter we will look at another sorting algorithm and will wish to compare with corresponding results of the new algorithm.

Exercise 3-5 Set C

There are 101 members in a youth symphony orchestra about to make a concert tour. A reporter asks the conductor, "What is the median age of your members?" He replies, "I have a list of the ages of the players. Will that help you?" (The median of an ordered set of numbers, you recall, is the middle number, if any.)

(a) Draw a flow chart to find the median age of the 101 players if the ages are taken in order from an alphabetical listing of the players. Can you solve this problem without using subscripted variables?

(b) If N is even, there is no middle number in a set of N numbers. How would you extend the idea of median to an ordered set of N numbers if N is even? Incorporate this idea into your flow chart which found the median of a set of N (odd) numbers. When revised your flow chart should output the median of any ordered set of N numbers (odd or even).

(c) Now the reporter, wanting to be prepared for the next orchestra to come to town, asks for a flow chart to give him the median age of any size group when the ages are given in arbitrary order. He would also like to know the ages of the oldest and youngest in the group. Prepare the flow chart.

3-6 Double Subscripts

Once you have mastered the use of subscripted variables in computing you will find that double subscripts offer very little additional difficulty.

In mathematics data often come to us in such a "rectangular array" of rows and columns as:

$$\begin{matrix} 5 & 2 & 7 & 1 \\ 9 & -4 & 0 & 2 \\ 6 & 7 & 3 & -2 \end{matrix}$$

Figure 3-28. Matrix

The mathematical term for such a rectangular array is "matrix." How often these matrices crop up you would have to see to believe.

One way in which such a matrix as the above might occur is as the "coefficient matrix" of a system of equations:

$$5W + 2X + 7Y = 1$$

$$9W - 4X + 0Y = 2$$

$$6W + 7X + 3Y = -2$$

The above matrix has three rows and four columns. Columns are vertical like the columns on a Greek temple. The individual numbers appearing in the array are called "entries." When you want to discuss the entry in a certain position, you can specify the position by giving the row and the column.

We see that a matrix is essentially a "table."

Column Row	1	2	3	4
1	5	2	7	1
2	9	-4	0	2
3	6	7	3	-2

Figure 3-29. The above matrix as a table

Double subscripts make their appearance when we introduce the notation used in talking about entries in a matrix. We use a variable with two subscripts

$$A_{I,J}$$

to indicate the entry in the I row and the J column. The row is always given first and the column second. Thus, if we let A represent the matrix at the beginning of this section as tabulated in Figure 3-29, then the value of $A_{2,3}$ is 0 while that of $A_{3,2}$ is 7.

As in the case of singly subscripted variables, we consider that there is a window box associated with each of the twelve variables $A_{1,1}$, $A_{1,2}$ and so forth as suggested in Figure 3-30.

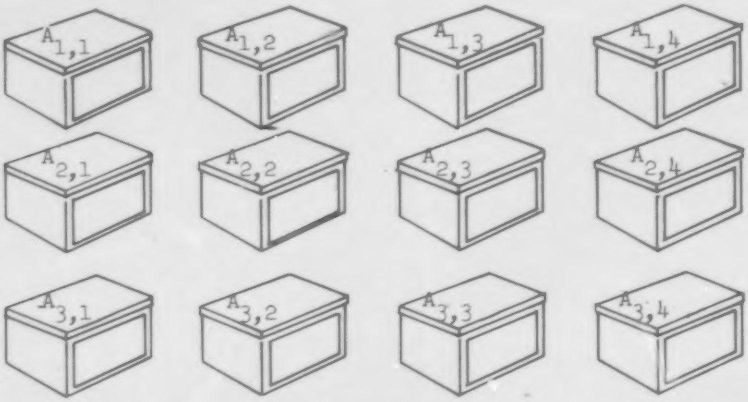


Figure 3-30. Window boxes for subscripted variables

If we wish to input a table into these window boxes, we could indicate this on a flow chart by the input box in Figure 3-31.

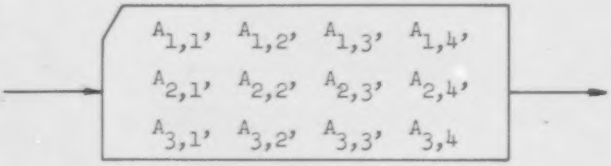

$A_{1,1}$, $A_{1,2}$, $A_{1,3}$, $A_{1,4}$,
$A_{2,1}$, $A_{2,2}$, $A_{2,3}$, $A_{2,4}$,
$A_{3,1}$, $A_{3,2}$, $A_{3,3}$, $A_{3,4}$

Figure 3-31. Input Box for Subscripted Variables

It would be good to have some notation (as in the last section) to refer to an entire matrix or to portions thereof. An extension of our previous notation is shown in Figure 3-32.

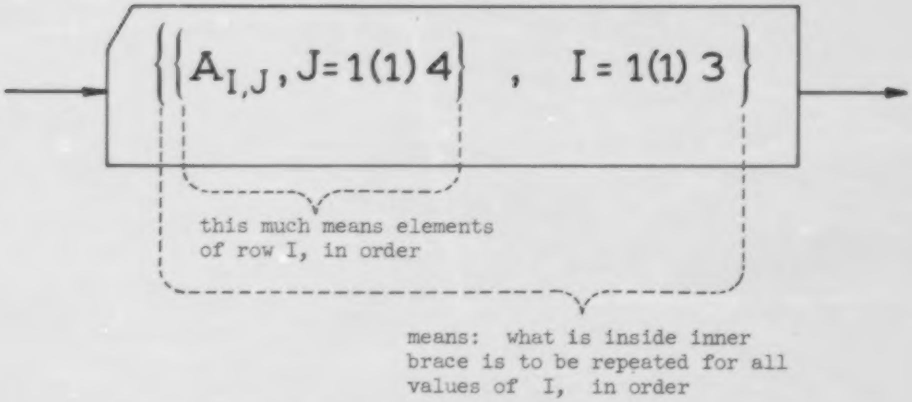

$\{\{A_{I,J}, J=1(1)4\}, I=1(1)3\}$
this much means elements of row I, in order
means: what is inside inner brace is to be repeated for all values of I, in order

Figure 3-32. Abbreviated input statement for doubly subscripted variables

This notation is an abbreviation for what appears in Figure 3-31. Mathematicians and computer programmers like to use such notation because it allows naming particular ordered subsets of matrix elements in an exact way. Thus, the way the braces are used in Figure 3-32 indicates that each row is read in completely (left to right) before going on to the next row. This could be important to know if the table is too large to put onto one card. We would then put each row (rather than each column) on a separate card. For our flow chart language, however, this information is quite superfluous. All we need to know is that an input box like that in Figure 3-32 will cause entries of a matrix like that in Figure 3-28 (or Figure 3-29) to be assigned to the appropriate variables represented in Figure 3-30.

Significant computations with doubly subscripted variables usually involve complicated looping and will, therefore, be left to the next chapter. We content ourselves here with a very simple example illustrating the use of double subscripts in flow charts.

Example: A zero sum game

We are given the matrix:

$$\begin{matrix} 6 & 2 & 5 & 4 & 3 & 1 \\ 9 & 0 & 8 & 3 & 2 & 6 \\ 1 & 8 & 5 & 4 & 1 & 1 \\ 8 & 3 & 7 & 3 & 6 & 3 \\ 5 & 5 & 4 & 8 & 1 & 2 \\ 3 & 2 & 1 & 6 & 4 & 8 \end{matrix}$$

We now describe a game employing this matrix. We have two dice, one green and one red. We roll the dice and let K denote the number on the green die and L that on the red die. Now we increase our score by the sum of the entries in the Kth row and we deduct from our score the sum of the entries in the Lth column. Can you see why this is called a "zero sum" game? Hint: Around what total score will the game hover after a large number of rolls of the dice?

We will construct a flow chart for this game. An outline of the steps involved in the problem is:

1. Input the given matrix.
2. Input values for K and L.
3. Calculate the sum of the entries in the Kth row.
4. Calculate the sum of entries in the Lth column.
5. Compute the difference of the values in Steps 3 and 4.
6. Print out this difference.

After a detailed analysis of Step 3, the flow chart should offer little difficulty. The analysis of this detail is given in Figure 3-33.

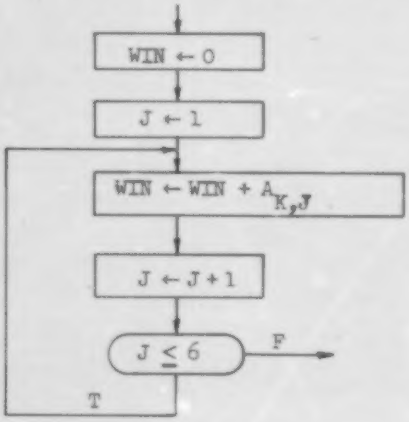


Figure 3-33. Detail of zero sum game

You should see that when we finally come out of this loop the value of WIN is the sum of the entries in the Kth row of the matrix. Notice that the value of K which determines the row in which entries are summed, remains the same during any one execution of the loop.

Now we exhibit the entire flow chart for this game in Figure 3-34.

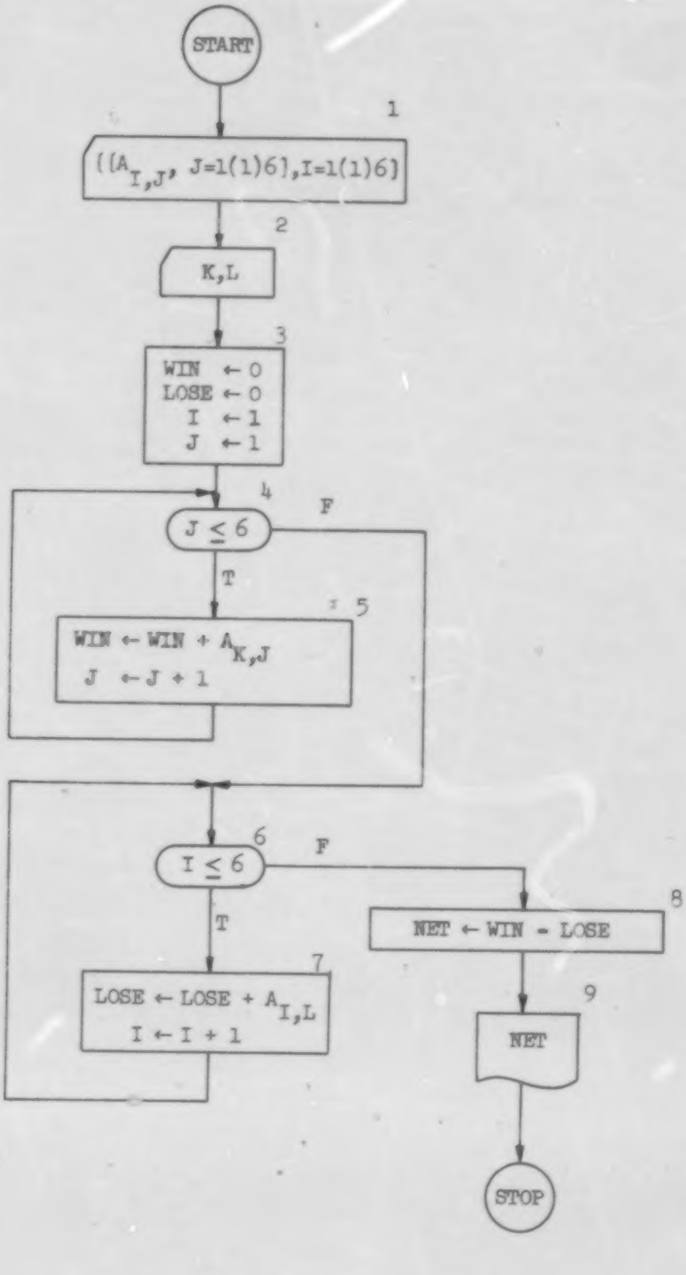

START
1
[[A_{I,J}, J=1(1)6],I=1(1)6]
2
K,L
3
WIN ← 0
LOSE ← 0
I ← 1
J ← 1
4
J ≤ 6
F
T
5
WIN ← WIN + A_{K,J}
J ← J + 1
6
I ≤ 6
F
T
7
LOSE ← LOSE + A_{I,L}
I ← I + 1
8
NET ← WIN - LOSE
9
NET
STOP

Figure 3-34. Flow chart for the game

It may be well to point out for contrast an alternative flow chart to Figure 3-34, which makes a sensible use of subscripted variable methods, for this problem and leads to a simpler program.

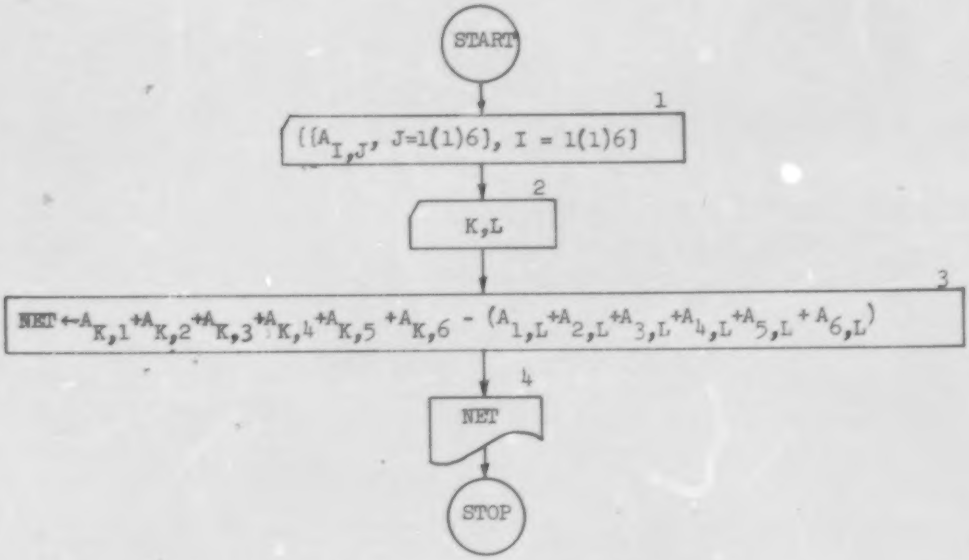

START
1
$\{\{A_{I,J}, J=1(1)6\}, I = 1(1)6\}$
2
K,L
3
$NET \leftarrow A_{K,1}+A_{K,2}+A_{K,3}+A_{K,4}+A_{K,5}+A_{K,6} - (A_{1,L}+A_{2,L}+A_{3,L}+A_{4,L}+A_{5,L}+A_{6,L})$
4
NET
STOP

Figure 3-35. Less instructive alternative

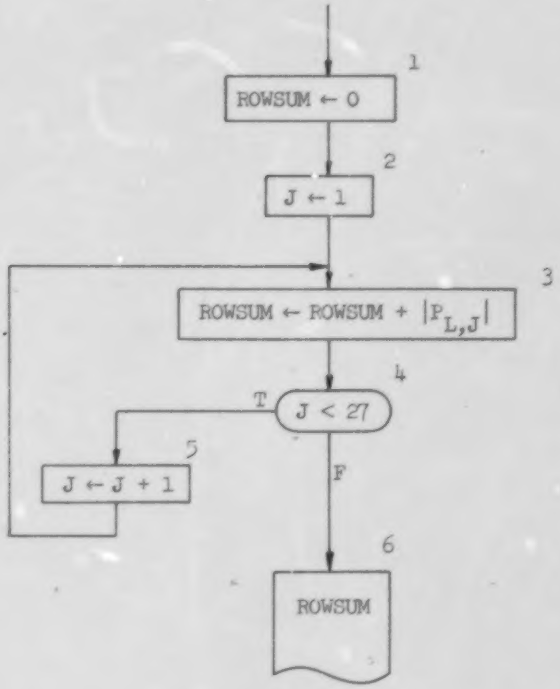
However, we lose some potential generality with this approach. Notice that in principle this game could also be played using larger matrices, say 8 × 8, 10 × 10, etc. Of course, for each new size we would need either dice with more faces, like octahedrons, or some other device for generating pairs of numbers. To generalize Figure 3-34 for any size array we need change only the 6's where they appear in Boxes 1, 4 and 6 to N and add a Box 0 to the flow chart at the start to read in this value of N--which could vary from game to game. Such generalization is not possible with the approach used in Figure 3-35. In short, while producing a shorter program, Figure 3-35 captures less of the spirit of our algorithmic method.

Exercises 3-6

In each of the following exercises, assume that values for the variables, or matrix entries which are mentioned are already assigned initial values. Your job is to flow chart the action described. (These are some of the elementary operations often performed with matrices. They are usually pieces of larger problems.)

For example, the matrix P has 22 rows and 27 columns. Find the sum of the absolute values of all entries in row L, where L has already been assigned a suitable value.

A suitable answer is:



1. For the same matrix P, find the sum of all but one of the entries in the Kth column. The exception is the entry in row 12 of that column. Call the sum being generated COLSUM.

2. For the same matrix P, add to each entry in row L the value of the corresponding entry (same column) of row M. As an actual example with a much smaller matrix, Q, we would have:

	before						after				
	3	4	2	5	-4		3	4	2	5	-4
Row L	3	9	1	2	-4	⟹	4	10	3	5	-2
Row M	1	1	2	3	2		1	1	2	3	2

3. For the same matrix P, add to each entry in row L, except in the Kth entry, 2 times the corresponding value of the Mth row.

4. For the same matrix P, interchange row L with row M.

5. For the same matrix P, find the entry in row L having the largest magnitude. Divide every entry in row L by the entry of largest magnitude.

Chapter 4

LOOPING

4-1 Looping

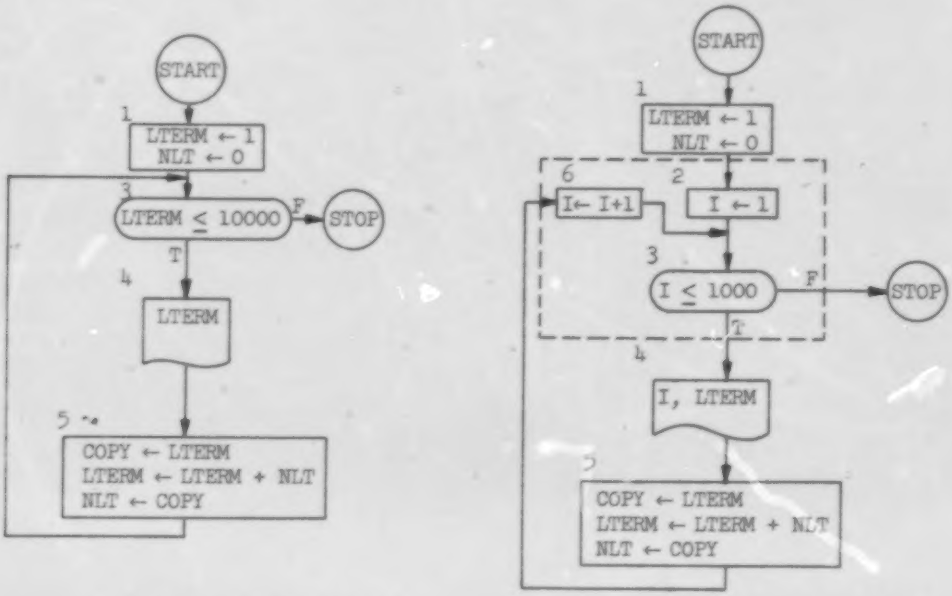
With the introduction of the concept of branching in Chapter 3 we have been able to develop some fairly complicated flow charts involving looping. "Looping" refers to the kind of "connections" which result in passing through the same box twice or many times during the course of a computation.

In this chapter we will study looping in more detail. Then we will develop a systematic way of treating one very important kind of looping.

We commence by putting down side-by-side in Figure 4-1 two different flow charts for the Fibonacci Sequence problem of Section 3-2. Remember that the Fibonacci Sequence,

1, 1, 2, 3, 5, 8, 13, 21, 34, ...,

has the property that each term (after the two 1's) is the sum of its two immediate predecessors.


START
1
LTERM ← 1
NLT ← 0
3
LTERM ≤ 10000
F
STOP
T
4
LTERM
5
COPY ← LTERM
LTERM ← LTERM + NLT
NLT ← COPY
START
1
LTERM ← 1
NLT ← 0
6
I← I+1
2
I ← 1
3
I ≤ 1000
F
STOP
T
4
I, LTERM
5
COPY ← LTERM
LTERM ← LTERM + NLT
NLT ← COPY

(a) No-Count Flow Chart (b) Ancestor of Iteration Box

Figure 4-1. Two Flow Charts for Fibonacci Sequence

The flow chart in Figure 4-1(a) represents an algorithm for computing and printing in order all terms of the Fibonacci Sequence which are less than 10000. The flow chart in Figure 4-1(b) represents an algorithm for computing and printing a numbered list of the first 1000 terms of the Fibonacci Sequence.

We can see that Box 5 is exactly the same in each flow chart. This box contains the fundamental computation in this algorithm.

Each flow chart has a loop, i.e., Boxes 3, 4, 5 in the first flow chart and Boxes 3, 4, 5, 6 in the second. These sequences of boxes are passed through (or "executed") over and over again. Each loop is equipped with an absolutely certain exit. In Figure 4-1(a) we exit or branch out of the loop as soon as the variable LTERM exceeds 10000. In Figure 4-1(b) we exit when I exceeds 1000. In Figure 4-1(b) the loop will be executed 1000 times. In Figure 4-1(a) it is not at all clear how many times the loop will be executed.

The reason that we can tell the number of times the loop will be executed in Figure 4-1(b) is that the loop is controlled by a counter, whereas this is not the case in Figure 4-1(a). The variable I works exactly like a counter.

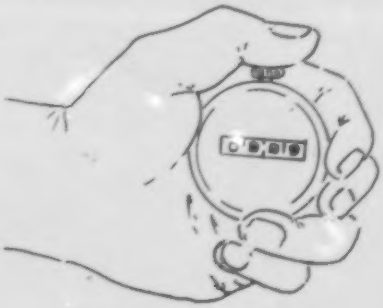


Figure 4-2. The Variable I

It is augmented, stepped-up, or incremented by 1 each time we pass through the loop. This is represented by Box 6 in Figure 4-1(b). Furthermore, Box 2 in this figure sets the counter to 1 at the start. Thus, the value of I gives us the number of transits through the loop we have made (including the one we are currently making).

In addition to acting as a counter, I has one additional duty; it controls the exit switch. When I counts up to 1000, it throws the switch allowing us to exit from the loop. Here we exit to a → STOP, but we could as well have gone to some other task. This "controlling" duty of the variable I is seen in Box 3.

To emphasize the distinction, we present still another flow chart, Figure 4-3, in which a counter has been added to Figure 4-1(a) to print out a numbered list.

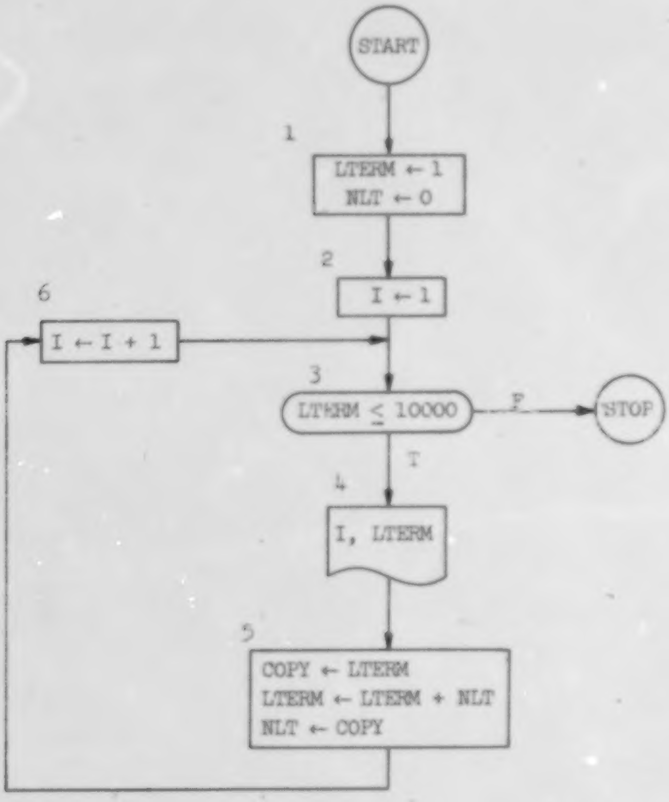

START
1
LTERM ← 1
NLT ← 0
2
I ← 1
6
I ← I + 1
3
LTERM ≤ 10000
F
STOP
T
4
I, LTERM
5
COPY ← LTERM
LTERM ← LTERM + NLT
NLT ← COPY

Figure 4-3. Non-Controlling Counter

We see that the variable I in Figure 4-3 has the same counting duty as the variable I in Figure 4-1(b), but it does not control the exit switch.

We see then that the variable I in Figure 4-1(b) has both counting and switching duties. You can conceive of I as a switchman who has been given the instructions, "Let the first 1000 through and then throw the switch."

The situation within the dashed lines of Figure 4-1(b) occurs so often that we introduce a special box to do the work of all three boxes.

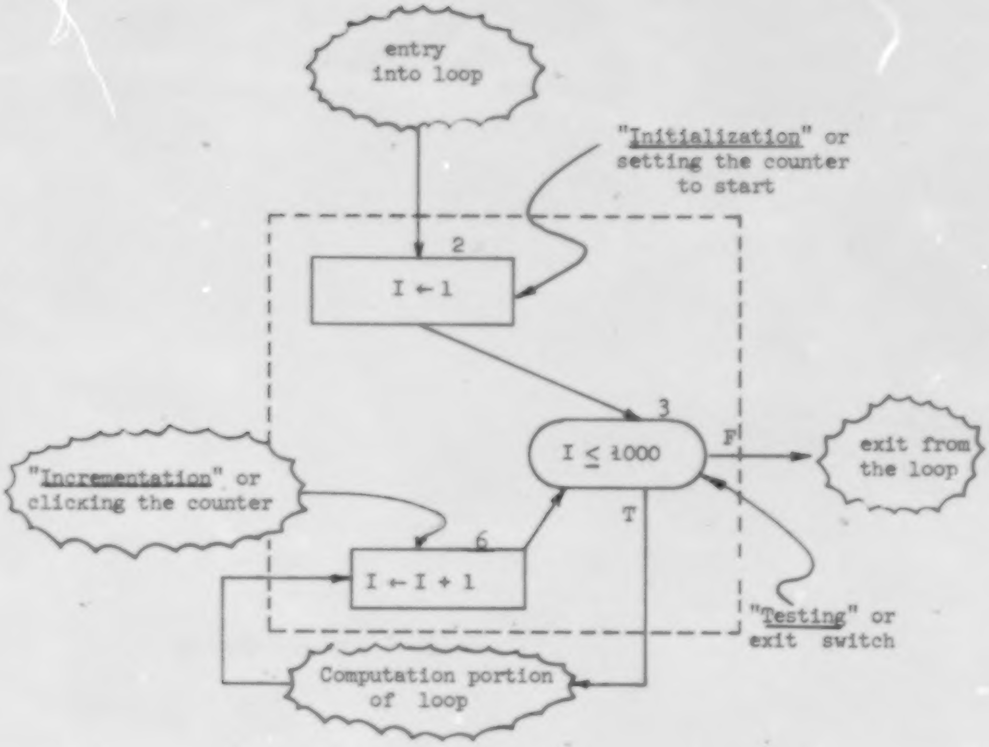

entry into loop
"Initialization" or setting the counter to start
2
I ← 1
3
I ≤ 1000
F
exit from the loop
T
"Incrementation" or clicking the counter
6
I ← I + 1
"Testing" or exit switch
Computation portion of loop

Figure 4-4. Embryo Iteration Box

And now!

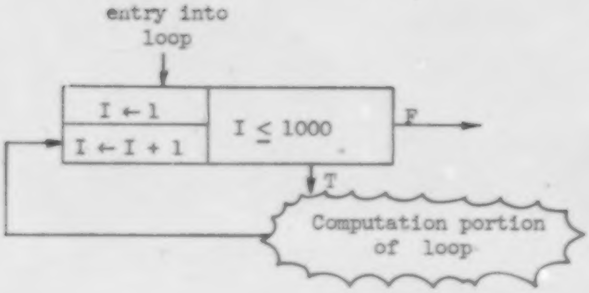
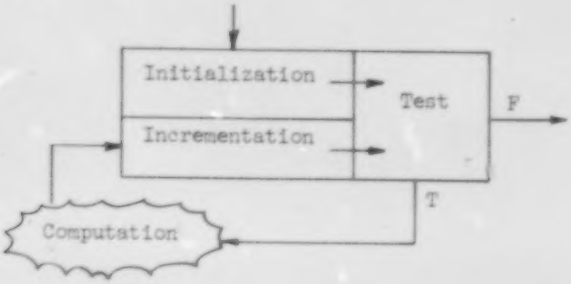

entry into loop
I ← 1
I ← I + 1
I ≤ 1000
F
T
Computation portion of loop

Figure 4-5. Birth of Iteration Box

The three-compartment box in Figure 4-5 is shorthand for the three boxes in Figure 4-4. Such a box can be used whenever a counter controls (the exit switch for) a loop. The exits from the two compartments on the left lead into the larger compartment on the right. We draw a schematic Iteration box to fix the names of the compartments.


Initialization
Incrementation
Test
F
T
Computation

Returning to our example of the Fibonacci Sequence we find that Figure 4-1(b) can be replaced by Figure 4-6.

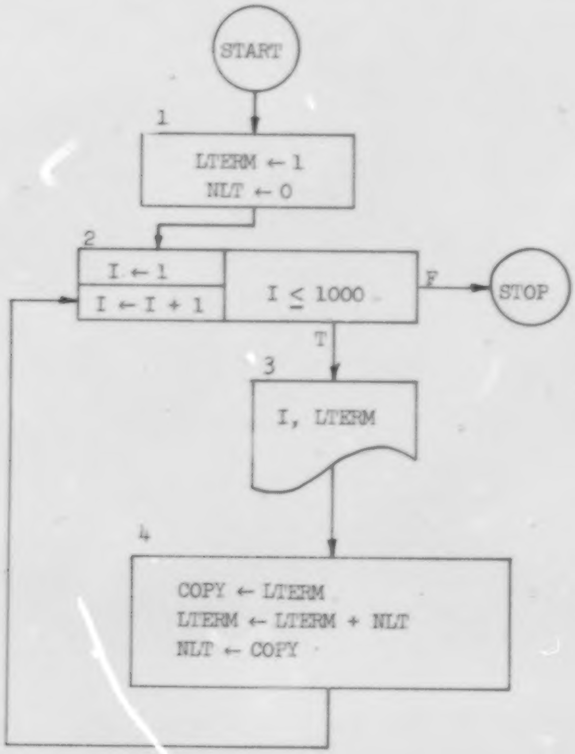

START
1
LTERM ← 1
NLT ← 0
2
I ← 1
I ← I + 1
I ≤ 1000
F
STOP
T
3
I, LTERM
4
COPY ← LTERM
LTERM ← LTERM + NLT
NLT ← COPY

Figure 4-6. Fibonacci Sequence Algorithm with Iteration Box

In Figure 4-6 Box 2 replaces Boxes 2, 3 and 6 of Figure 4-1(b). If you have understood what it is that an iteration box does, then Figure 4-6 should be easier to read than Figure 4-1(b). We'll soon see that iteration boxes make flow charts easier to write, too. For, whenever we realize (or even suspect) that we have a loop controlled by a counter we draw the iteration box and try to hang the loop on it.

We must remember that the heart of the loop is in the computation portion. The iteration box merely represents the in and out mechanism. However, such a flow chart as the following is possible.

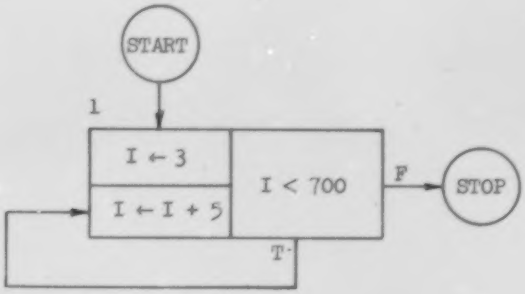
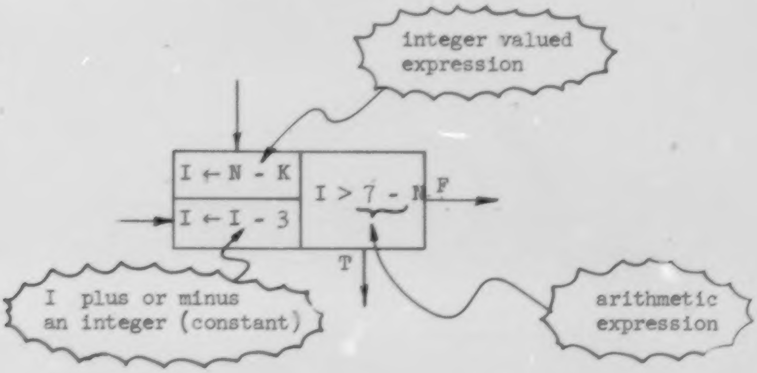

START
1
I ← 3
I ← I + 5
I < 700
F
STOP
T

Figure 4-7. The "Little Dandy" flow chart

The best that can be said of this "algorithm" is that having no output, it saves paper. This flow chart does show us that we can initiate with integers other than 1, increment with integers other than 1 and exit on integers other than 1000. We give below a diagram showing the most general forms of iteration box generally used in this course.


integer valued expression
I ← N - K
I ← I - 3
I > 7 - N
F
T
I plus or minus an integer (constant)
arithmetic expression

Exercises 4-1

1. The instructor who gave our first problem for computing and printing

$$D = \sqrt{A^2 + B^2 + C^2}$$

now restates the problem this way: "You are given 50 different sets of data consisting of four items each: An identification number ID, A, B, and C. Produce a printed table having five columns of numbers: ID, A, B, C, and the computed value of D. Each printed line in the table is to correspond to one input data set and the computed value of D." Your job in this exercise is to draw a flow chart for this computation. Use an iteration box. Only four flow chart boxes are needed (not counting START and STOP).

2. Redraw the flow chart you made in Problem 1 to achieve the following two improvements in the algorithm simultaneously.

   (a) Handle N sets of data where N is any integer (within reason).

   (b) After printing the table, print out a message "END OF TABLE" and return to accept another group of N data sets where N may now have a different value.

3. In Figure 4-6 we studied a way to generate the terms of the Fibonacci Sequence. Now you are to flowchart a related algorithm: Generate both the Fibonacci Sequence and its sum sequence. Let F_I be the Ith term of the Fibonacci Sequence. Thus, F_3 is 2, F_4 is 3, F_5 is 5, etc. Let S_I be the sum of all terms of the Fibonacci Sequence up to and including the Ith term.

$$S_5 = 1 + 1 + 2 + 3 + 5 = 12$$

$$S_6 = 1 + 1 + 2 + 3 + 5 + 8 = 20, \text{ etc.}$$

Each term of the S-sequence is a cumulative sum. Your flow chart should generate pairs of values of these two sequences and print the pairs as they are generated. The first pair to be printed is F_3, S_1, and the second is F_4, S_2, etc. The algorithm should terminate after printing 60 such pairs.

As an added challenge, see if you can write the flow chart without using subscripts.

4. Recall Figure 3-24(b) which was an algorithm for computing the points won or lost in one spin of the carnival wheel (i.e., for one data pair S and M). This algorithm allowed us to have an arbitrary point rule by input of four values into vector elements P_1, P_2, P_3, P_4.

   Now suppose we are interested in determining our score after a large number, say N, of arbitrarily chosen data pairs S, M. For the moment we won't concern ourselves with where these data pairs came from. Your job is to revise the flow chart in Figure 3-24(b) so that it now does the following:

   (a) inputs values of P_i for a 4-point rule;

   (b) determines a point value for each of N data pairs S, M and, instead of printing these values, forms their sum in (SUM);

   (c) after the N data pairs have been "processed", prints N and SUM using appropriate literals to explain the significance of the values that are printed. For example, "After 35 spins your score is 552 points."

   If you can't fight your way through this exercise just yet--postpone it until after you have done several of the exercises in Set A of the next section.

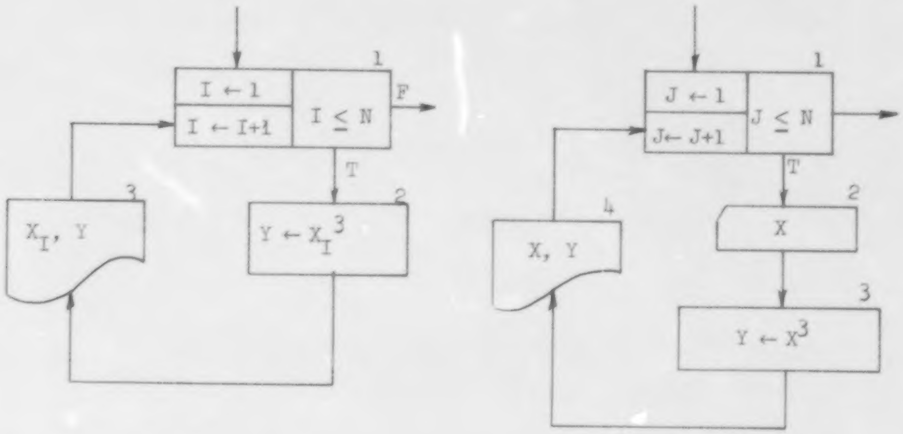
5. Simplified Model of Payroll Computation.

   The workers in a plant are assigned numbers from 1 to N. Let T_i be the number of hours worked by worker number i and let R_i be his hourly rate of pay. The payroll department wishes to input the time data from the Time-keepers' Department and the rate figures from the Personnel Department and output the weekly wages for each worker and the total payroll. Draw a flow chart to do this job. You may assume that the Timekeepers' data comes in the form of an ordered list of the T_i's from 1 to N and that Personnels' data is a second ordered list of the R_i's from 1 to N.

4-2 Illustrative Examples

In this section we wish to present a portfolio of examples illustrating the iteration box. We present them at a rather brisk pace and they will get gradually more complicated. In order to make what we wish to emphasize stand out, we will usually present only fragments of flow charts. Consider the task: Given N numbers, print out these given numbers and their cubes, thus in effect constructing a table of cubes.

Suppose the list is already stored in the computer's memory, in locations belonging to a subscripted variable, X_I. Then we will run through the subscripts reading out of memory the values of the X_I, and making the desired computations. This suggests the use of an iteration box with the loop variable doing the running. This is shown in 4-8(a). In Figure 4-8(b) the same computation is performed with the data stored on cards rather than in the machine.



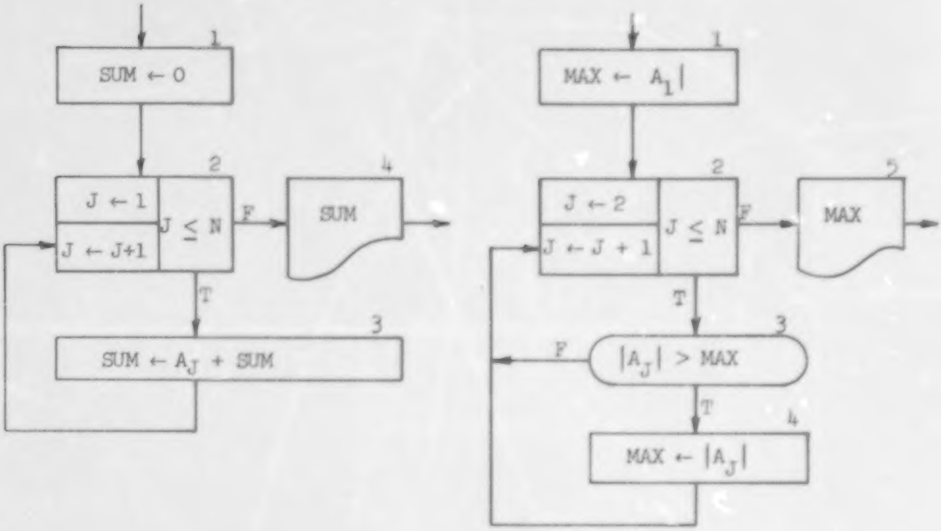
(a) From internally stored data

(b) From external data

Figure 4-8. Making a table of cubes

Notice that in Figure 4-8(b) the loop variable is nowhere to be seen in the computation portion of the loop. Figure 4-8(a) provides our first experience with a common occurrence, that of a loop variable going click, click, click through the subscripts of a subscripted variable.

We see this again, in fact, in our next two examples: Adding up a list of numbers already stored in memory, and, second, finding from a list of numbers in memory, the maximum of their absolute values. These are flowcharted in Figure 4-9. If you have trouble understanding this loop, review Figures 3-24(b) and 3-25 where we first discussed an algorithm like this one.



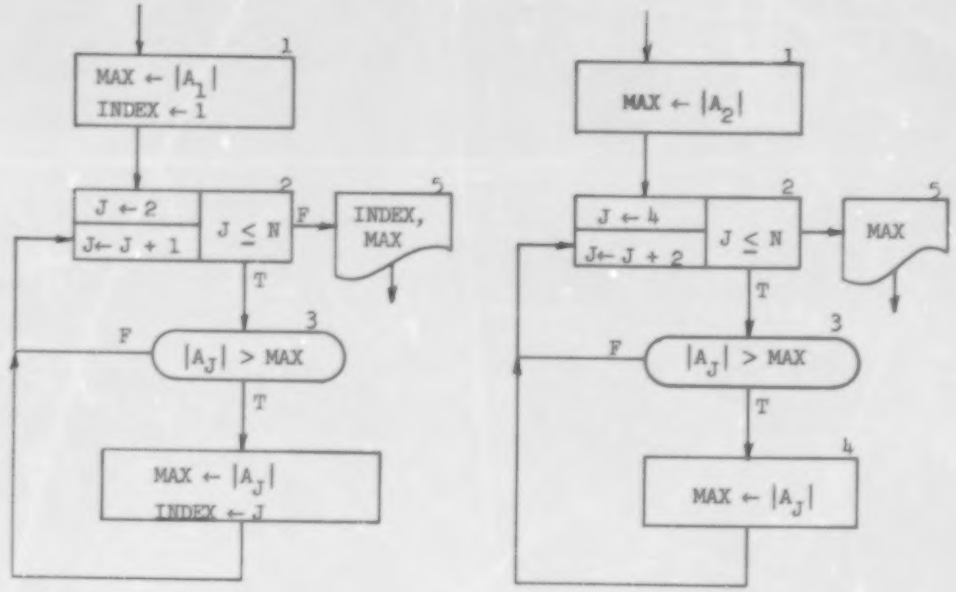
(a) The sum of the components of a vector

(b) The maximum of the absolute values of the components of a vector

Figure 4-9. Application of iteration boxes to calculation with vectors (subscripted variables)

There is a fundamental difference between the algorithm of Figure 4-9 and those of Figure 4-8. In any transit of the loops of Figure 4-8 no use is made of calculations made in previous transits, while that is certainly not the case in Figure 4-9.

We find in Figure 4-10 two variants of Figure 4-9(b). The first shows the modification of Figure 4-9 which must be made if we wish to print out the value of J for which $|A_J|$ is the maximum. The second shows the algorithm for finding the maximum value of $|A_J|$, but over only the even values of J.

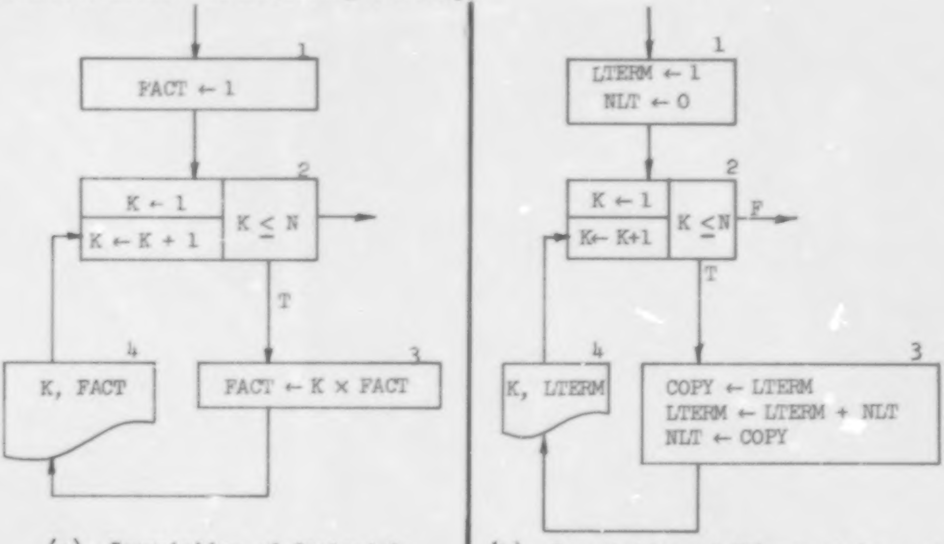

1
MAX ← |A1|
INDEX ← 1
2
J ← 2
J← J + 1
J ≤ N
F
5
INDEX, MAX
T
3
F
|AJ| > MAX
T
MAX ← |AJ|
INDEX ← J
1
MAX ← |A2|
2
J ← 4
J← J + 2
J ≤ N
5
MAX
T
3
F
|AJ| > MAX
T
4
MAX ← |AJ|

(a) Outputting index (b) Maximizing over even subscripts

Figure 4-10. Two variations of Figure 4-9(b)

Notice the counter we used in (b) of Figure 4-10 counts by 2's and not by 1's.

The first of the loops of Figure 4-11 exhibits the calculation of factorials and the second is borrowed from the Fibonacci Sequence algorithm of Figure 4-6. These algorithms share the property that no data is injected (stored or external) after first entering the loop.


1
FACT ← 1
2
K ← 1
K ← K + 1
K ≤ N
T
4
K, FACT
3
FACT ← K × FACT
1
LTERM ← 1
NLT ← 0
2
K ← 1
K← K+1
K ≤N
F
T
4
K, LTERM
3
COPY ← LTERM
LTERM ← LTERM + NLT
NLT ← COPY

(a) Computation of factorials (b) Computation of Fibonacci Sequence

Figure 4-11. Loops without data

Exercises 4-2 Set A

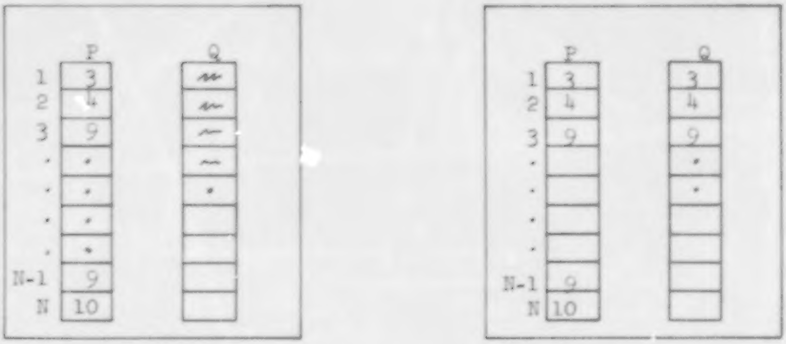
You have in storage two columns of N numbers each. One column is called P, the other Q. For each problem in this set your job is to convert the word statement to an equivalent partial flow chart. You should find the iteration box helpful. You may wish to first flow chart the computation part of the loop, then hang it from the proper iteration box, and finally precede the iteration box, if appropriate, by an initializing box.

1. Think of the I^{th} value of P and the I^{th} value of Q as the pair P_I, Q_I. Interchange the values in every such pair.

2. Modify the flow chart drawn for Problem 1 so only even-indexed pairs are interchanged. Does it matter whether N is even or odd?

3. Modify the flow chart drawn for Problem 1, assuming you wished to interchange only every third pair of values beginning with the fifth pair.

4. Move the first $[N/2]$ elements of the vector P to the vector Q. See picture (Figure 4-12).



Picture of memory before "move"

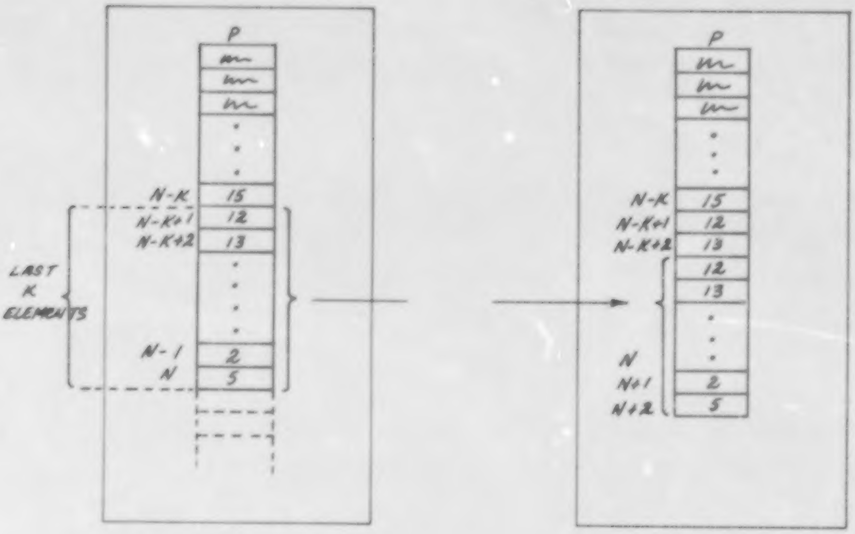
Picture of memory after "move"

Figure 4-12. Moving elements of a vector

5. Move the last $[N/2]$ elements of the vector P to the first $[N/2]$ positions of vector Q. Assume N is even. Hint: What is the index of the first element of P which is to be moved?

6. Same problem as Exercise 5--but don't assume N is even.

7. Let each of the last K elements of the N-element vector P be "shifted" or moved two positions in memory to make room for the later insertion of two new values at positions N - K + 1 and N - K + 2. See Figure 4-13.



Picture of Memory

Before move

Picture of Memory

After move

Figure 4-13. Shifting elements of a vector

8. You have already stored 100 input values for elements P_1, P_2, ..., P_{100}.

   (a) Form the sum of their cubes (in SUMCUB).

   (b) Form the sum of the negative values (in SUMNEG).

   (c) Form the sum SUMCUB, SUMNEG, and SUMBIG (where SUMBIG is the sum of the absolute values greater than 50 in magnitude).

9. Refer to the flow chart you constructed in Problem 1, Exercises 3-6. Redraw the flow chart using an iteration box (sum of entries in the K^{th} column of the matrix P except for element in Row 12.)

10. Refer to the flow chart you constructed for Exercise 2, Section 3-6. Redraw the flow chart using an iteration box (replacing entries of the L^{th} row of matrix P by the sum of Row L and Row M entries).

11. Refer to the flow chart you constructed for Exercise 3, Section 3-6. Redraw the flow chart using an iteration box (replacing entries of L^{th} row by sum consisting of Row L entry and 2 × Row M entry, but leaving Column K entry of Row L unchanged).

In each of the following two exercises assume there are N values currently assigned to the P vector in memory.

12. Search the list in the forward direction, i.e., P_1, P_2, P_3, etc., for the first value greater than 50 in magnitude, assigning this value to W and 1 to ANY. If no such value is found, assign 0 to ANY. In either case now proceed to the same point in the flow chart.

13. Search the list in the backward direction, i.e., P_N, P_{N-1}, P_{N-2}, etc., for the first value greater than 50 in magnitude, assigning this value to W. If no such value is found, assign 50 to W. In either case, now proceed to a common point in the flow chart.

14. Search the N elements of the P vector for the non-zero element of largest magnitude less than $|M|$. Assume the value of M has already been stored in memory. Assign the value of the vector element found in this search to T. If no such value is found, print the message "NONE" and stop.

15. Search all N elements of the P vector for the element which is the largest in value and is still less than the value currently assigned to M. Assign the value found to T. If none is found, print "NONE" and stop.

In the following two exercises assume that all entries of a matrix Q are stored in memory. Q has M rows and N columns.

16. Search the L^{th} row of Q for the smallest value. Assign this value to SMALL.

17. Search the R^{th} column backwards (i.e., from bottom to top) for the first entry, if any, that is at least as large as the current value of T. Assign the row value for this entry to ROW and the value itself to BIG. If no such value is found assign the value zero to ROW.

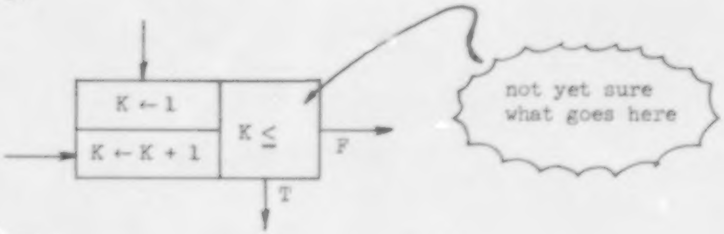
We have now seen a number of examples illustrating the use of iteration boxes. But how, in the course of drawing a flow chart, can we tell whether an iteration box will be useful? The answer is that we will want to use an iteration box whenever we have a loop controlled by a counter. Whenever this situation exists (or when we strongly suspect that it does) we draw the iteration box and try to hang a loop on it. We may draw the iteration box before knowing everything that goes inside it.

As an example, consider the problem of finding all the integer factors of a given integer N. If N is large, this task is very tedious, as you will know if you have ever tried it. We will be very glad, therefore, to have a computer do the job for us.

The word statement of the algorithm for this problem is very simple:

> Go through the integers, 1, 2, 3, 4, etc., checking each one to determine whether it is a divisor of N, and if it is, write it down.

Now for the flow chart. Since for each integer we must check whether it divides N, we have a repetitive process, a loop. Because we perform the calculation for 1, 2, 3, 4, etc., it would seem that our loop is controlled by a counter. Now we draw our iteration box, noting that we are not yet sure where to stop.



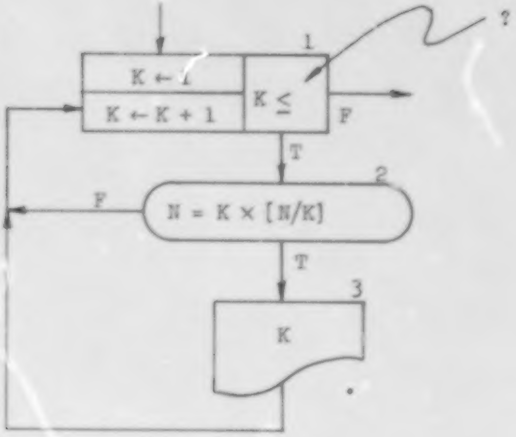
Of what does our calculation consist? Of determining whether K is a divisor of N. But how shall we express this question? Well, for K to be a divisor of N means that N/K is an integer or, equivalently, $[N/K] = N/K$. Thus,

$$K \text{ is a divisor of } N$$

is equivalent to

$$N = K \times [N/K].$$

We put this in our flow chart!



Now we come to the important question of where to stop our computation. We could go all the way to N, i.e., we could put N in the empty space in the test compartment of the iteration box. Then if N were a million, we would have to go through the loop a million times. Must we do that? At this point a look at the mathematics of the situation will help.

Whenever we find one integer factor of N, say K, we have really found two, because N/K is also an integer factor. Moreover, these two factors cannot both be less than $\sqrt{N}$, else we would have

$$N = K \times (N/K) < \sqrt{N} \times \sqrt{N} = N$$

which is a contradiction. By the same reasoning, K and N/K are not both greater than $\sqrt{N}$. If they were, we would have a similar contradiction

$$N = K \times (N/K) > \sqrt{N} \times \sqrt{N} = N.$$

Thus, whenever we express N as the product of two factors, one of these factors is $\leq \sqrt{N}$ and the other is $\geq \sqrt{N}$. This means that we only have to go as far as $\sqrt{N}$ in our search for factors if, in our output step, we print out the value of N/K along with each factor K.

Our complete flow chart is then seen in Figure 4-14.

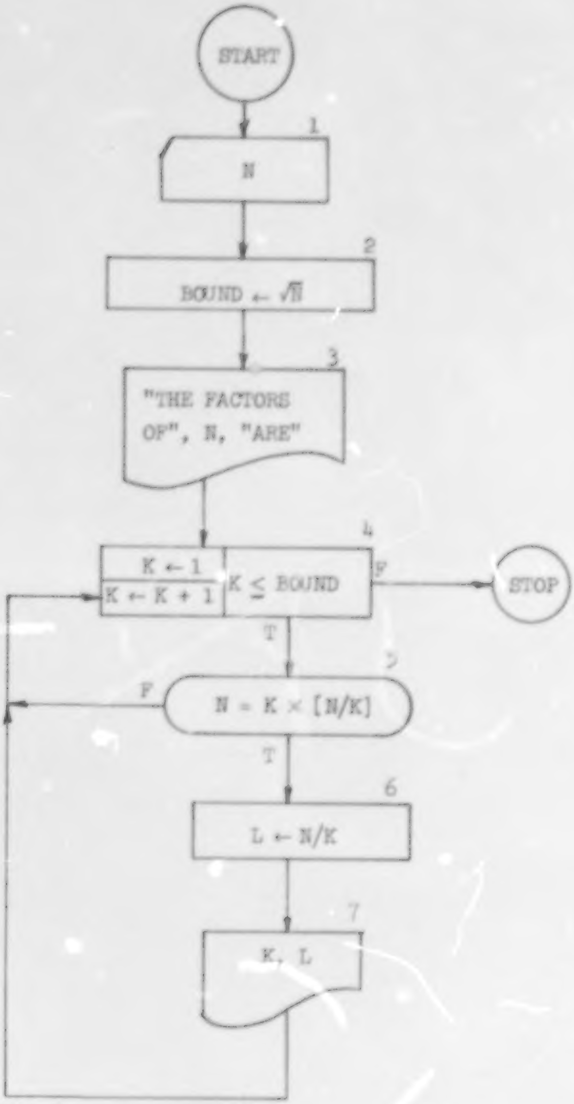

START
1
N
2
BOUND ← √N
3
"THE FACTORS OF", N, "ARE"
4
K ← 1
K ← K + 1
K ≤ BOUND
F
STOP
T
5
F
N = K × [N/K]
T
6
L ← N/K
7
K, L

Figure 4-14. Finding the factors of N

We see that if N were 1,000,000 we would now pass through the loop only 1000 times. Quite a saving over our original plan to pass through the loop a million times!

The last example of this section is the problem of evaluating a polynomial. We will illustrate our methods with a third degree polynomial and then generalize to N^{th} degree. Consider the expression:

$$A_0 + A_1 \times X + A_2 \times X^2 + A_3 \times X^3.$$

After values are assigned to the components of the vector A, this expression represents a polynomial of degree no greater than 3. Next, a value is assigned to X; the polynomial can be evaluated.

We first describe the usual method of performing this evaluation. We evaluate each term in the order written, adding this value to a cumulative sum of the terms computed so far. We also keep the last power (PWRX) of X computed to simplify the computation of the next higher power. This process is obviously controlled by a counter which runs through the subscripts of A. But should we initiate the loop variable at 0 or at 1 ? There doesn't seem to be much computation at 0. Our thoughts so far are shown in Figure 4-15.

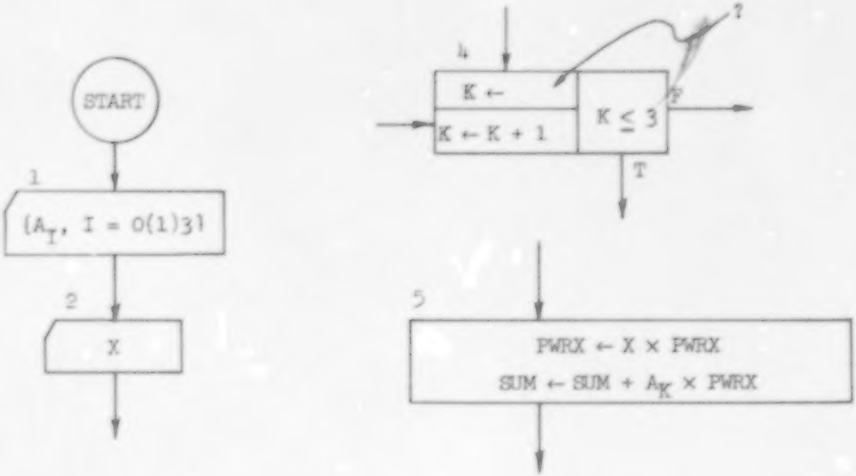


Figure 4-15. Pieces of polynomial evaluation

Box 5 in Figure 4-15 contains the entire loop calculation. Our decision about what initial values to assign to SUM and PWRX decides the question of how to initialize K in Box 4. Now we can draw our flow chart (Fig. 4-16).

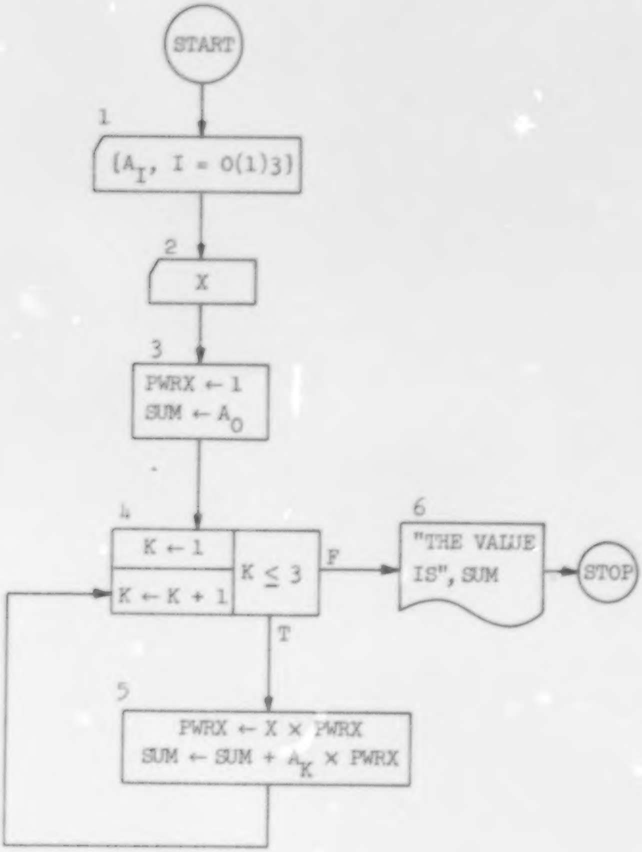

START
1
(A_I, I = 0(1)3)
2
X
3
PWRX ← 1
SUM ← A_0
4
K ← 1
K ← K + 1
K ≤ 3
F
T
6
"THE VALUE IS", SUM
STOP
5
PWRX ← X × PWRX
SUM ← SUM + A_K × PWRX

Figure 4-16. Evaluation of polynomial, everyday method

It should be obvious how we can generalize this flow chart to work for polynomials of arbitrary degree. We have only to replace the occurrences of "3" in Boxes 1 and 4 by "N", and input N, either in Box 1 or ahead of that box. Instead of stopping we could, of course, go back to get another value of X or to get another polynomial.

Now that we have solved the problem we ask (as usual), "Is there another way to do it?" There is, in fact, another way, and a most elegant one. We take a polynomial

$$B_0 \times X^3 + B_1 \times X^2 + B_2 \times X + B_3$$

and express it in the following way.

$$((B_0 \times X + B_1) \times X + B_2) \times X + B_3$$

You will have to satisfy yourselves that these two expressions are equivalent. The second of these expressions is in very inconvenient form for all mathematical purposes except evaluation.

In constructing the flow chart, at each step the variable VALUE represents the number that X is being multiplied by. You might enjoy trying to draw the flow chart yourself before looking at the solution in Figure 4-17.

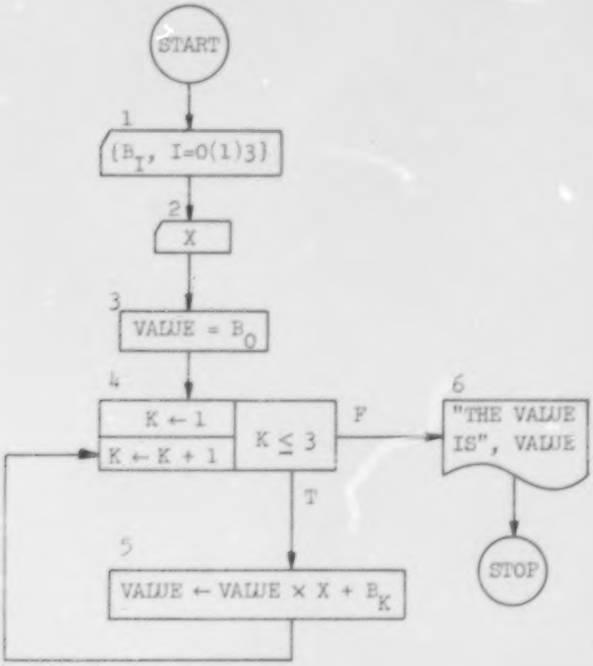

START
1
{B_I, I=0(1)3}
2
X
3
VALUE = B_0
4
K ← 1
K ← K + 1
K ≤ 3
F
T
5
VALUE ← VALUE × X + B_K
6
"THE VALUE IS", VALUE
STOP

Figure 4-17. Evaluation of Polynomial, Sunday method

Again, of course, we can generalize to degree N by replacing the occurrences of "3" in Boxes 1 and 4 by "N" and inputting N prior to (or in) Box 1.

Now let us compare the two algorithms. Certainly, the simplicity of the computation in Box 5 of Figure 4-17 appeals to our esthetic sense, but in the last analysis the key question is, "Which algorithm uses the least computer time?"

To answer this, compare the computation portion (Box 5) of the two loops. In the first method there will be two multiplications and one addition for each transit of the loop. In the second method there will be one multiplication and one addition in each transit of the loop. The second (Sunday) method is therefore shorter. If the polynomial were of degree N, it would save N multiplications. And if you intend to use this algorithm to evaluate many polynomials, then the second method will save a number of multiplications equal to the sum of the degrees of all the polynomials to be evaluated.

Exercises 4-2 Set B

1. Given a set of N values of a vector X, i.e., $X_1, X_2, \ldots, X_N$, and given a value A,

   (a) draw a flow chart for the computation of NUM defined mathematically as an N-term product:

   $$NUM = (X_1 - A) \times (X_2 - A) \times (X_3 - A) \times \ldots \times (X_N - A)$$

   Show input and output of all required data and results.

   (b) Same as (a) except that NUM has the K^{th} one of the N terms omitted.

2. Suppose in the preceding exercise you are told that the given value of A is equal to X_K. What data value, required as input in 1(b), is no longer needed? Redraw the flow chart to display the computation of DEN, which is defined the same as NUM, except $X_K = A$ and the term $(X_K - A)$ is omitted.

3. (a) Preliminary. In Problem 4, Exercise 4-1, you were asked to spin the carnival wheel N times. Did you finish this exercise? If not, do so now before going on to the main task described in the next paragraph.

(b) Main. We begin now to develop more seriously the concept of simulating the playing of a game for the purpose of predicting, with the aid of a computer, something about its outcome. It's a "spinning wheel game". Imagine you are given an inexhaustible supply of data pairs, s, m, for input. These data are somehow representative of what a person might actually experience if he were to take turns spinning the wheel with one or more other players. We will say that a "game" consists of a series of spins for a given player and terminates whenever the magnitude of his score, |SUM|, exceeds some given critical value, CV. We shall say that the "length" of the game is the number of spins in the series. The question we really want to ask is: How many turns "on the average" can a player be "expected" to take before CV $\leq$ |SUM| ?

In this exercise you are preparing the groundwork to answer the question later. Your job now is to draw a flow chart that simulates one complete game. The paragraph below contains some guidelines--to be consulted only after you have experimented with a plan of your own.

(1) As in the earlier exercise, first input the four values constituting the "point rule" to be used.

(2) Next input a value for CV, the critical value.

(3) Then input a series of data pairs, s, m.

(4) After each data pair the new value for the net winnings (SUM) is computed and a counter L of the number of spins is updated.

(5) Whenever the absolute value of SUM exceeds the CV, we print the values of L, CV, and SUM and then stop.

(6) For insurance against an endless loop, we print out an error message and stop if ever L exceeds 1000.

Remember to use an iteration box where you think it can help to keep the flow chart simple in structure.

4-3 Table-Look-Up

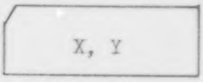
Now we begin an algorithmic investigation into the subject of table-look-up; the looking up values of a tabulated function, such as when we "go to the tables" to find the value of sin(.3217) or of $\sqrt{147.62}$.

Example 1. Table-look-up by matching

Our first example of table-look-up does not even involve an iteration box. We have a function, F, and as in common mathematical notation we write

$$Y = F(X) .$$

Now we have a stack of cards, each card punched with a value of the variable X and the corresponding value of Y.



This means that each card represents an ordered pair of numbers (X,Y), related by Y = F(X). No two cards then can have the same value of X punched on them unless the values of Y are also the same. This is what is meant by saying that "Y is a function of X". The stack of cards can be regarded as a table of values of the function, F.

Now, to look up the functional value of a certain number, one method would be to go through the cards comparing this number with the value of X on each card, and, when equality is found, to print the corresponding value of Y. Figure 4-18 is a flow chart for this process.

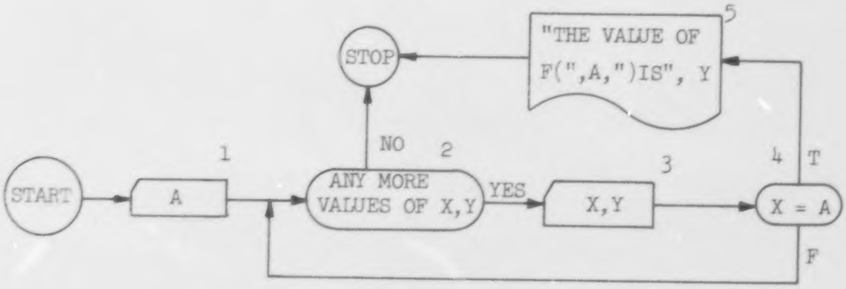
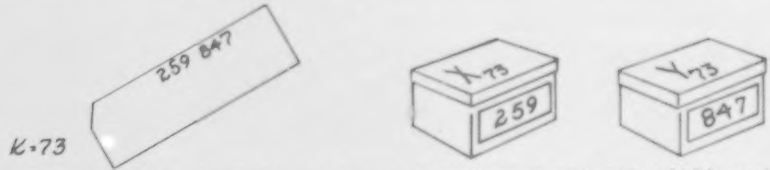


Figure 4-18. Primitive Table-Look-Up

Any flow chart to read a whole table into storage will use subscripted variables. We suppose that we have a stack of cards with two numbers X_K, Y_K punched on each, and with X_K and Y_K satisfying

$$Y_K = F(X_K) .$$

Notice that the subscript changes each time a card is read. Data from different cards goes into different window boxes. If 1000 cards are to be read, then 2000 window boxes must be made available to receive the data on them.



For our flow chart language we introduce a slightly different type of input box.

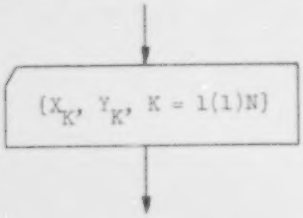
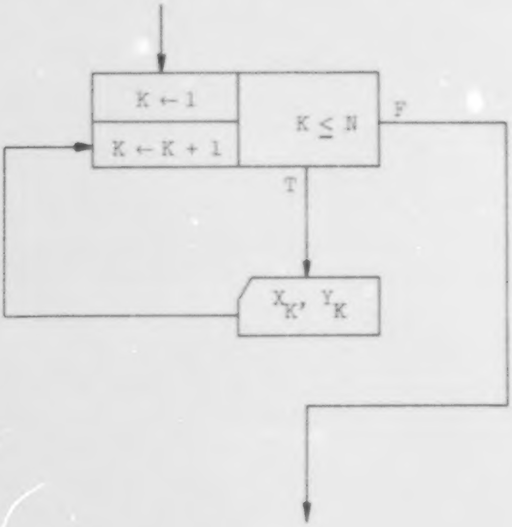


Figure 4-19. An input step for a stack of N ordered pairs of subscripted variables

Though we have not seen an input instruction quite like this before, it is clear that its effect is equivalent to the following combination of boxes:



Once the table has been read in, we select a value for which F is sought. We compare this value with each of the X_I until we find equality; then we print out the value of Y_I. The flow chart is seen in Figure 4-20.

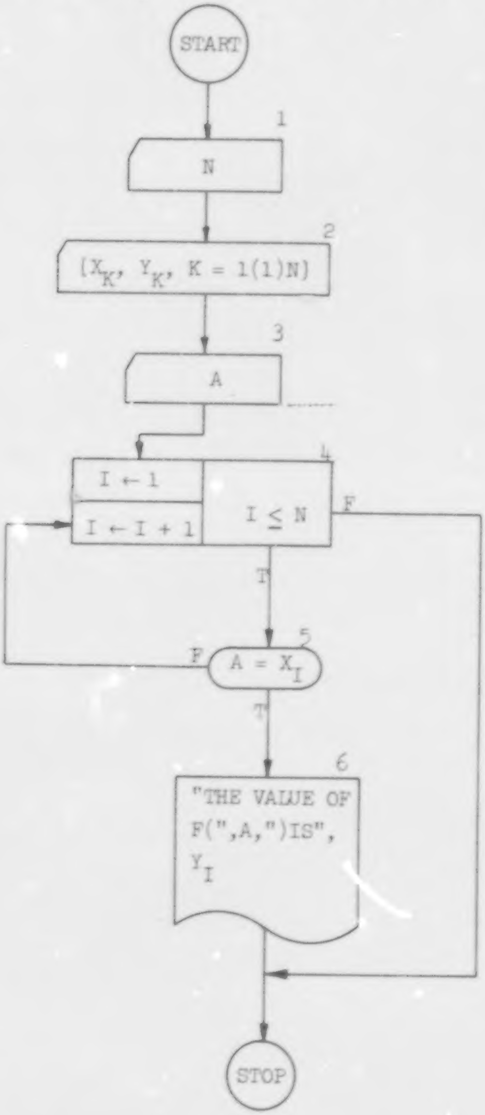

START
1
N
2
(X_K, Y_K, K = 1(1)N)
3
A
4
I ← 1
I ← I + 1
I ≤ N
F
T
5
F
A = X_I
T
6
"THE VALUE OF F(",A,")IS", Y_I
STOP

Figure 4-20. Look-up from internally stored table

What is important here is the way in which the iteration box helped us to draw the flow chart. Notice, however, that there is a second exit from the loop besides the one from the iteration box.

In Figure 4-20 we could have gone back for more values of A instead of stopping.

Example 2. Table-look-up: Bracketing box entries in an ordered set

Have you ever had to look up values in a table? Suppose you are given a value of X, and you want to find the value of sin(X). What happens, of course, is that you don't find your value of X listed. So you note the nearest listed values above and below, and write these down together with their functional values, as in Figure 4-21.

your given value of X → .5836

X	sin(X)
⋮	⋮
.5760	.5446
.5789	.5471
.5818	.5495
.5847	.5519
.5876	.5876

(a) What you see in the table

.5818	.5495
.5836	?
.5847	.5519

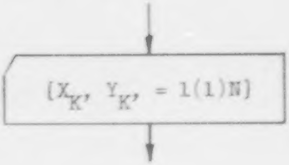
(b) What you write down

Figure 4-21. Reading a table

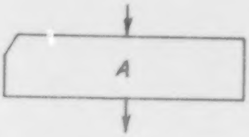
Now you usually "interpolate" a value between the two tabulated values of sin(X) in the same proportion as that interpolated between the two tabulated values of X.

We will construct a flow chart for instructing a computer to do everything except the final interpolation. It would be easy to instruct the machine to perform this step, too, but we want to focus our attention on the table-look-up problem. We will print as output the number whose functional value we wish to find, the closest tabulated values of X, above and below, and the corresponding values of Y.

We will assume in this problem that the values of X are arranged (indexed) in increasing order. We input the table



and the value we are looking up in the table



Our task involves comparing A with successive values of X. Again we have a loop that can be controlled by an iteration box. The computation consists merely of comparing A and X_I and printing out the desired information when we bracket A between two tabulated values of X. Of course, if we hit A on the nose we output that information, too. The flow chart in Figure 4-22 is self-explanatory.

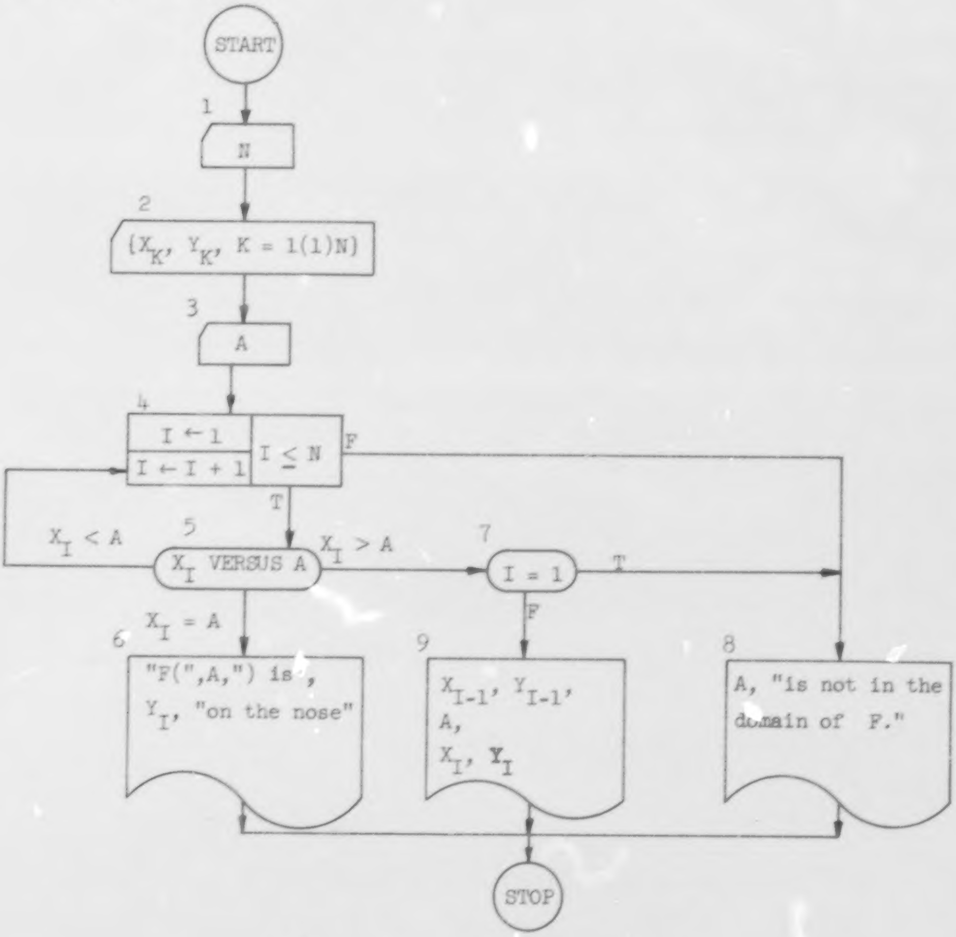

START
1
N
2
$\{X_K, Y_K, K = 1(1)N\}$
3
A
4
$I \leftarrow 1$
$I \leftarrow I + 1$
$I \leq N$
F
T
$X_I < A$
5
X_I VERSUS A
$X_I > A$
7
I = 1
T
F
$X_I = A$
6
"F(",A,") is",
Y_I, "on the nose"
9
X_{I-1}, Y_{I-1},
A,
X_I, Y_I
8
A, "is not in the domain of F."
STOP

Figure 4-22. Simple scan table-look-up in an ordered set of values

Exercise 4-3 Set A

Improve the flow chart in Figure 4-22 so that in place of Box 9 we will print the interpolated value of Y (call it YINT) along with the value of A. Hint: Figure 4-22a should help you to see how YINT may be computed.

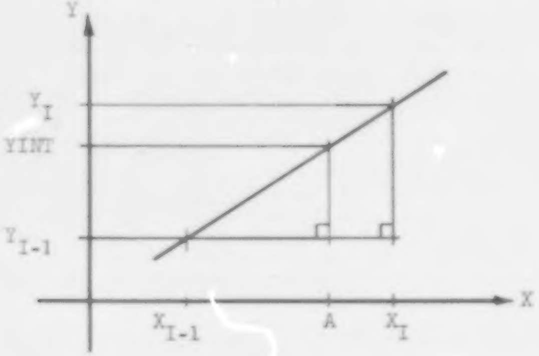
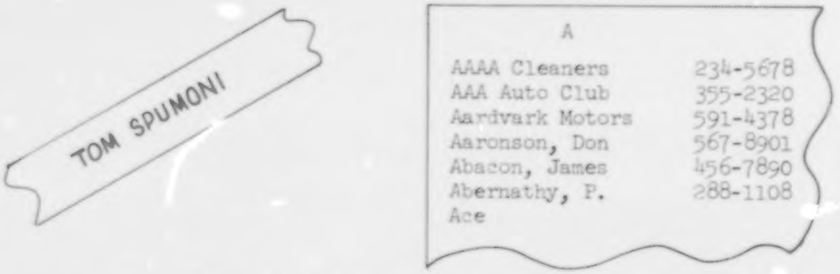


Figure 4-22a. Illustration of straight line interpolation

[†]Example 3. Table-look-up - bisection method

And now we ask the same old question. Can we improve on the algorithm? Let's compare the algorithm with what we do in real life. In the algorithm we take a value of A and start at the beginning of the table and compare with each entry. Is this what we do in real life? Take the analogy of a telephone book. We want to find the number of Tom Spumoni. Do we start at the beginning (Figure 4-23) comparing Spumoni with each entry?



A

AAAA Cleaners	234-5678
AAA Auto Club	355-2320
Aardvark Motors	591-4378
Aaronson, Don	567-8901
Abacon, James	456-7890
Abernathy, P.	288-1108
Ace	

Figure 4-23. In search of Spumoni

[†]If time is short, the rest of Section 4-3 may be skipped without loss of continuity.

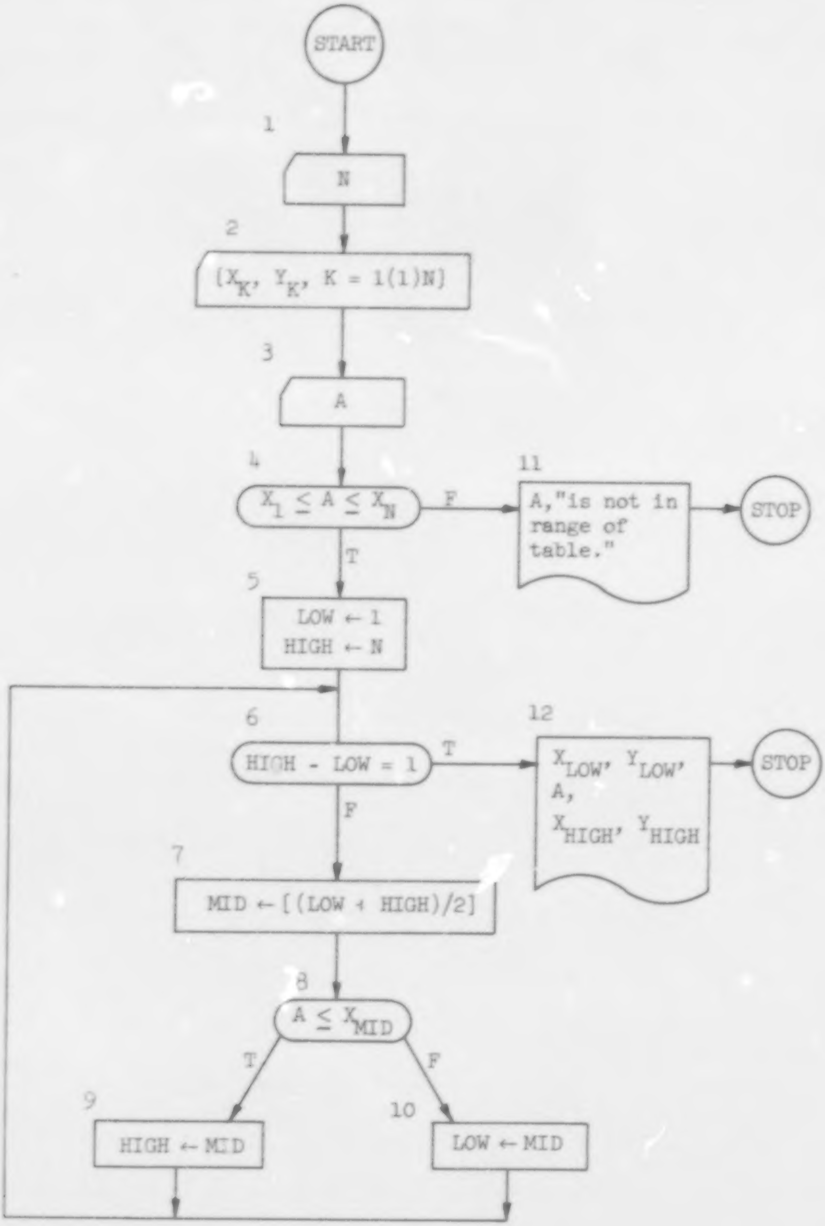
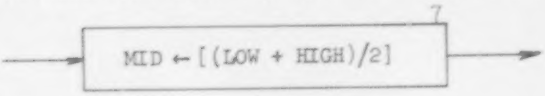

START
1
N
2
[X_K, Y_K, K = 1(1)N]
3
A
4
X_1 ≤ A ≤ X_N
F
11
A,"is not in range of table."
STOP
T
5
LOW ← 1
HIGH ← N
6
HIGH - LOW = 1
T
12
X_LOW, Y_LOW, A, X_HIGH, Y_HIGH
STOP
F
7
MID ← [(LOW + HIGH)/2]
8
A ≤ X_MID
T
F
9
HIGH ← MID
10
LOW ← MID

Figure 4-24. Table-look-up using bisection technique
(Highly instructive)

Certainly not! What we do is split the book in the middle and check to see which "half" the name is in. Then we split that "half" and so on. Just as this is a faster way for you to look things up, so it is for a computer.

The flow chart for this algorithm is presented in Figure 4-24. We use two auxiliary variables, LOW and HIGH, to indicate the lower and upper indices of the part of the table we are currently confined to.

On each loop we find the midpoint of LOW and HIGH, that is



and test to see whether

$$A \leq X_{MID} .$$

If so, MID becomes the new HIGH (Box 9) and if not, MID becomes the new LOW (Box 10). Box 4 determines at the outset whether A is in the range of the table. Box 6 is the stopping mechanism. When $HI - LO = 1$ we know that A is bracketed between two table entries with consecutive subscripts, i.e., that

$$X_{LO} \leq A \leq X_{HI} .$$

The computation portion of the loop (Boxes 6 through 10) exhibits the "bisection technique". Study this computation until you are sure you understand it. The idea occurs over and over again in computing and often represents maximum efficiency. You'll see bisection again in Chapter 7.

It is interesting to compare the efficiency of the algorithms in Figures 4-22 and 4-24. The loop of Figure 4-22 (Boxes 4 and 5) will be passed through $N/2$ times on the average. The loop in Figure 4-24 will be executed a number of times equal to or one less than the number of digits required to express N in the binary system. For example, if N is 1,000,000, Figure 4-22 requires 19 or 20 transits since $2^{19} < 10^{6} < 2^{20}$.

There is no iteration box in the flow chart of Figure 4-24. The reason is that the loop in this algorithm is not controlled by a counter. There is a valuable lesson here: You should not try to force algorithms into iteration box form when it seems difficult to do so. Iteration boxes are not useful in all loops. They are useful only when the loop is controlled by a simple counter.

Exercise 4-3 Set B

You may have noticed that the algorithm in Figure 4-24 lacks one feature exhibited by the one in Figure 4-22. That is, in the latter, if an exact match is found, a message like

"F(24.2) is 39.25 on the nose"

is printed.

Your job in this exercise is to redraw Figure 4-24,or whatever portion is necessary, with the "on the nose" feature added.

[††]Example 4. Table-look-up in an unordered set of values

Suppose that the values of X_I had not been indexed in increasing order. This sort of thing might happen if the table were constructed out of empirical data collected by a number of investigators.

There are two different plans we could follow:

1. We could look up our value in the table as it stands.
2. We could first sort the data according to increasing values of X_I and then look up in the sorted table.

The first plan would be followed only to look up a very small number of values in the table. When many values are to be looked up, the second plan is much shorter.

If we sort the data first, the sorting process would be followed by a bisection look-up algorithm as in Figure 4-24. Sorting has already been discussed in Section 3-5 and will be discussed further in Section 4-4. We turn our attention now to the first plan.

Our goal here is the same as in the algorithm of Figure 4-22. to squeeze A as tightly as possible between two tabulated values of X_I,

$$X_{LO} \leq A \leq X_{HI} .$$

The difference here is that we will know that we have attained our result, only after we have scanned the entire table. Furthermore, no bisection technique can be used here since the values of X_I are not indexed in any order.

[††] If time is short, this section can be skipped without loss of continuity.

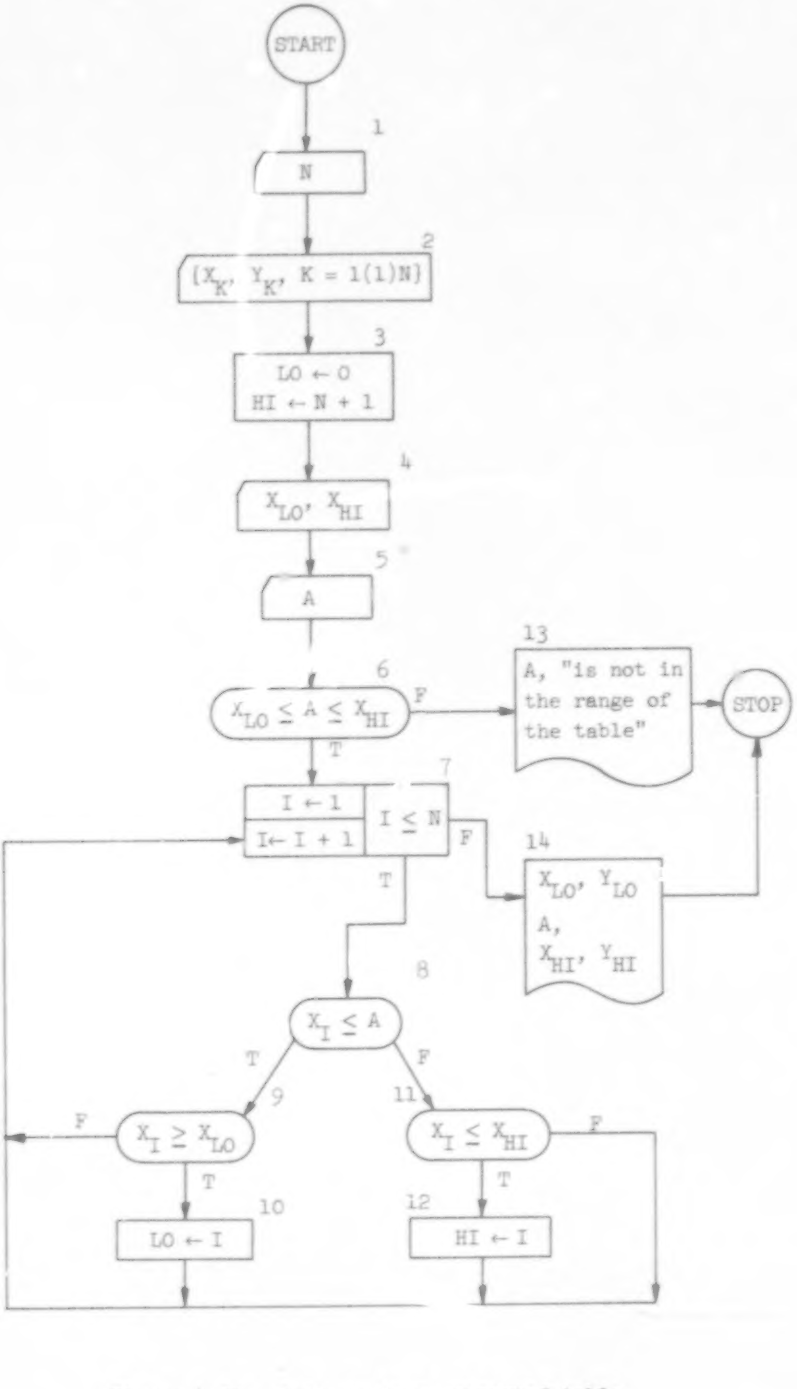

START
1
N
2
$\{X_{K}, Y_{K}, K = 1(1)N\}$
3
LO ← 0
HI ← N + 1
4
X_{LO}, X_{HI}
5
A
6
$X_{LO} \leq A \leq X_{HI}$
F
T
13
A, "is not in the range of the table"
STOP
7
I ← 1
I ← I + 1
I ≤ N
F
T
14
X_{LO}, Y_{LO}
A,
X_{HI}, Y_{HI}
8
$X_I \leq A$
T
F
9
$X_I \geq X_{LO}$
F
T
10
LO ← I
11
$X_I \leq X_{HI}$
F
T
12
HI ← I

Figure 4-25. Look-up in an unsorted table

In constructing this algorithm it is assumed that the maximum and minimum values of X_I are known and are input as X_{LO} and X_{HI}. Now we know

$$X_{LO} \leq A \leq X_{HI} .$$

We proceed to scan the table replacing the value of LO with that of I whenever we find

$$X_{LO} \leq X_I \leq A$$

Similarly, we replace HI by I whenever $A \leq X_I \leq X_{HI}$. Since we are scanning the entire table we clearly have a loop controlled by a counter and hence an iteration box. The algorithm is seen in Figure 4-25.

4-4 Nested Loops

By th[illegible]ested loop" we refer to algori[illegible]ich have, like the silhouette shown in Figure 4-27, a loop within a loop. In this silhouette Boxes 3 through 7 constitute a loop while Boxes 4 and 5 form an inside loop. Remember, in a flow chart, whenever an arrow goes back to a box already passed through, then you have a loop.

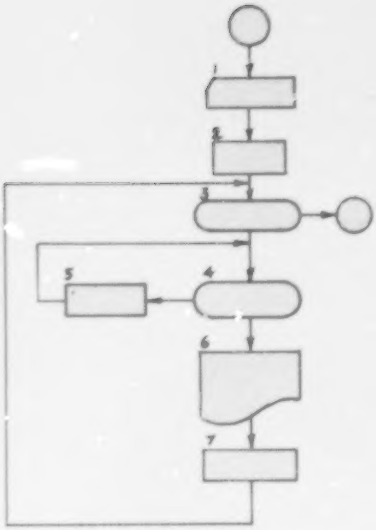
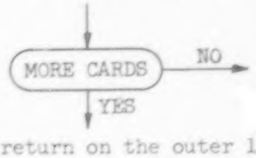


Figure 4-27. Silhouette of a nested loop

You have already seen numerous examples of nested loops of a rather trivial kind in which the "return" on the "outer" loop merely involved coming back for more data as in the next silhouette (Figure 4-28). Here the inner loop consists of Boxes 3, 4 and 5. Box 3 is an iteration box. The outer loop consists of Boxes 1, 2, 3, 4 [illegible] 5. Clearly Box 1 is supposed to be



and the return on the outer loop is merely for the purpose of coming back to repeat the same calculation with new sets of data.

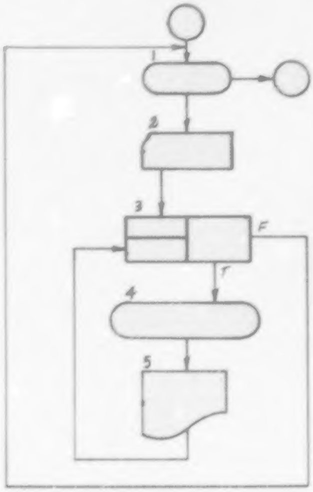
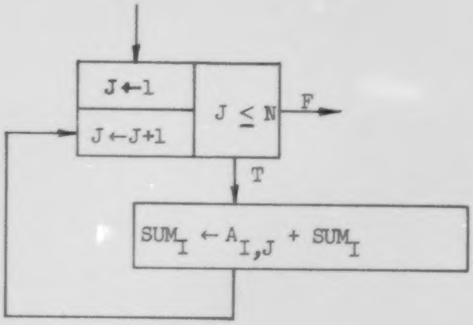


Figure 4-28. Silhouette showing nesting formed when returning for more data

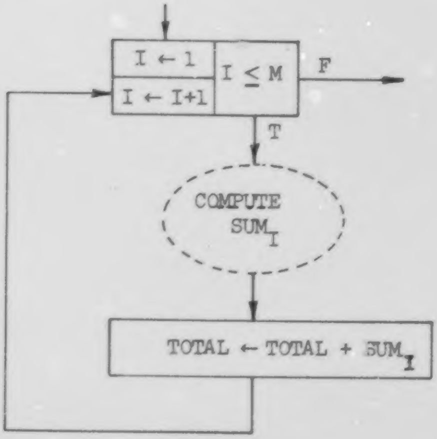
The sorting algorithm of Section 3-5 had a non-trivial use of a nested loop although we did not call your attention to it at the time. It is quite possible to construct valid algorithms containing nested loops without being conscious of the existence of this nesting. But when both the inner and outer loops are controlled by iteration boxes we will usually be conscious of the nesting. The most natural example of this is some systematic processing of the entries in a matrix. Suppose we wish to find the sum of all entries in a matrix. First we add up each row, and then add the resulting sums.

In this example we can mentally separate the calculations in the two loops. The inner loop, adding up the entries in a given row, say the I^{th}, would be given by



J ← 1
J ← J+1
J ≤ N
F
T
$SUM_I \leftarrow A_{I,J} + SUM_I$

And then the outer loop



I ← 1
I ← I+1
I ≤ M
F
T
COMPUTE SUM_I
$TOTAL \leftarrow TOTAL + SUM_I$

The dotted line is to be filled in with the inner loop. There is nothing left to do but to input the matrix, initialize the various SUM's and TOTAL to zero and output the final answer. The flow chart is Figure 4-29. The return arrow for the inner loop goes from Box 7 to Box 6, while that for the outer loop goes from Box 9 to Box 4.

Many computations that involve matrices have similar flow charts. Several will be seen in the next set of problems.

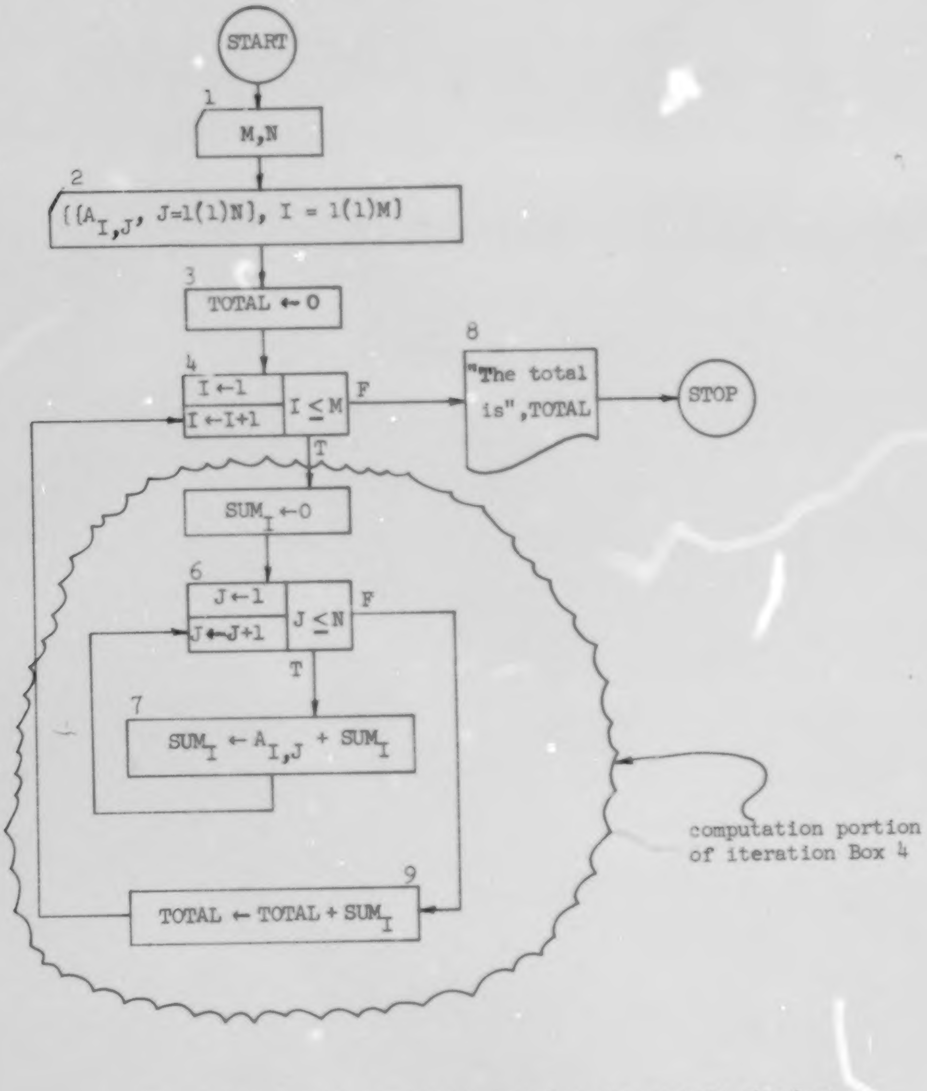

START
1
M,N
2
{{$A_{I,J}$, J=1(1)N}, I = 1(1)M}
3
TOTAL ← 0
4
I ← 1
I ← I+1
I ≤ M
F
T
8
"The total is",TOTAL
STOP
SUM_I ← 0
6
J ← 1
J ← J+1
J ≤ N
F
T
7
SUM_I ← $A_{I,J}$ + SUM_I
9
TOTAL ← TOTAL + SUM_I
computation portion of iteration Box 4

Figure 4-29. The sum of the entries of a matrix showing nested iteration boxes

Exercises 4-4 Set A

In the following exercises you are to draw a flow chart equivalent to each word problem. Each involves a nested loop, and you will find iteration boxes helpful. Assume in each case that the matrix P, having M rows and N columns, is already stored in memory.

1. Search P for the element of largest absolute value. Assign this element to BIG and print the value of BIG. Hint: Start by assuming the entry of largest magnitude is zero.

2. Search P for the element of largest value (not absolute value), assigning it to LARGE. Print the value of LARGE and the row number ROW, and column number, COL, where this value was found. Hint: Start by assuming the largest value is $P_{1,1}$.

3. Search for the least non-zero element in odd-numbered rows and even-numbered columns, and assign its value to LEAST. While conducting this search, keep a tally of the number of zeros found, ZTALY, and then print values for LEAST and ZTALY. If all elements are zero, the value printed for LEAST should be zero.

4. Add a multiple, T, of the first row entries to the entries of all other rows of P. For example, if T = 2, we show the action on a 4-row by 4-column matrix P.

$$\underbrace{\begin{matrix}1&2&1&1\\3&4&2&5\\1&2&1&2\\3&1&3&2\end{matrix}}_{\substack{\text{before}\\\text{action}}}\qquad\qquad\underbrace{\begin{matrix}1&2&1&1\\5&8&4&7\\3&6&3&4\\5&5&5&4\end{matrix}}_{\substack{\text{after adding}\\2\times\text{ row 1 to}\\\text{each of the}\\\text{other rows}}}$$

5. Determine the minimum value in each column, MIN, of the matrix P and print it out with its row and column identification, ROW and COL. If there is more than one occurrence of the minimum value, report the last one found. For the 4 × 4 array shown in the preceding exercise in the "before" state, the desired output for this exercise would be:

```
MIN    ROW    COL

 1      3      1
 1      4      2
 1      3      3
 1      1      4
```

6. A matrix which has the same number of rows and columns is called a "square matrix". In the next three exercises we shall assume that P is square (M rows and M columns). The "main diagonal" of a square matrix is a line of entries each having equal row and column subscripts, i.e., $P_{1,1}$, $P_{2,2}$, $P_{3,3}$, etc. The i^{th} element on the main diagonal can therefore be referred to as $P_{i,i}$.

Assign to SUM1 the sum of all entries to the <u>left of the main</u> diagonal of the square matrix P, accumulating the terms row by row. Hint: Make yourself a picture of the triangular group of entries that fall in the category to be summed. What is the first row involved? What is the last column involved? For any row to the left of the main diagonal, what are the subscripts of the rightmost entry?

7. Form the sum of all entries which are situated to the <u>right of the main</u> diagonal of P, accumulating the terms row-by-row. (Refer to Exercise 6.)

8. The "triangle" to the left of the main diagonal which you worked with in Exercise 6 is often called the "lower triangle" and the one to the right of the main diagonal is often called the "upper triangle". In this exercise we wish to search the upper triangle column-by-column starting from the <u>last</u> column. We will search each column from top to bottom for the first entry that is at least twice as large in magnitude as its immediate predecessor in the same column. An entry which exhibits this increased magnitude will be termed a PIG.

A PIG can occur in any but the leftmost column of the upper triangle. Print all values of PIG as they are discovered along with their row and column subscripts I and J. If no PIG is found, print "NONE". What is the smallest matrix which can have a PIG? (Answer: a 3 × 3.) Hint: To search a column for a PIG can the top element be one? What is the row subscript for the bottommost element in the J^{th} column?

The Stickler Example

Next we give a little problem to help drive home the power of a computer. You will recognize the problem as being of a type often encountered in algebra courses (and puzzle books).

STICKLER: Find all the three-digit numbers which are equal to the sum of the cubes of their digits.

The reason such a problem emphasizes the power of a computer is that this problem, though possible, is extremely tedious to do by hand calculation. However, it is a trivial problem for a computer and the algorithm is absurdly easy to write.

If we let the digits of the number be H, T, and U, then the three-digit number is

$$100 \times H + 10 \times T + U.$$

The problem then is to find and print out all the triples of digits (H, T, U) which satisfy the condition:

$$100 \times H + 10 \times T + U = H^3 + T^3 + U^3 .$$

Consequently, we would expect to find in our flow chart the structure shown below.

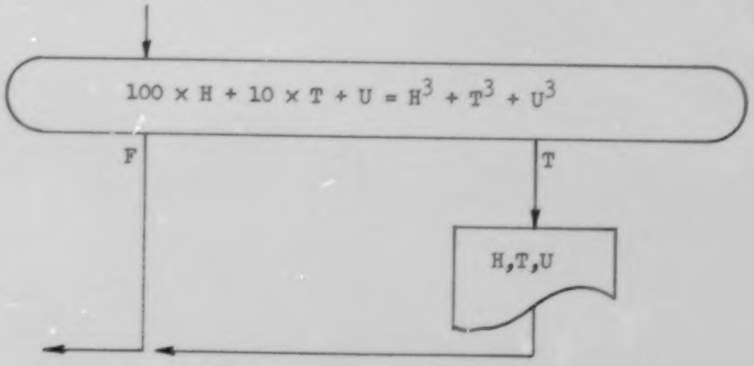


Figure 4-30. Computation for the Stickler

This is, in fact, the only computation performed in this algorithm. The rest of the problem merely makes the various values of H, T, and U available for the test of Figure 4-30. The process of making these values available involves nested loops. This process can be described somewhat vaguely in words as: When a value is assigned to H we then let T "run through" the digits from 0 to 9 and when values are assigned to both H and T we then let U "run through" the digits from 0 to 9. In this explanation we are trying to explain briefly the process of counting as performed by the odometer on a car where we consider each rotating wheel of the odometer as a variable and the value showing as the value of that variable.

This commonplace idea becomes even clearer in the flow chart for the algorithm given in Figure 4-31. The initial value of H is 1 rather than zero because we are looking for three-digit numbers.

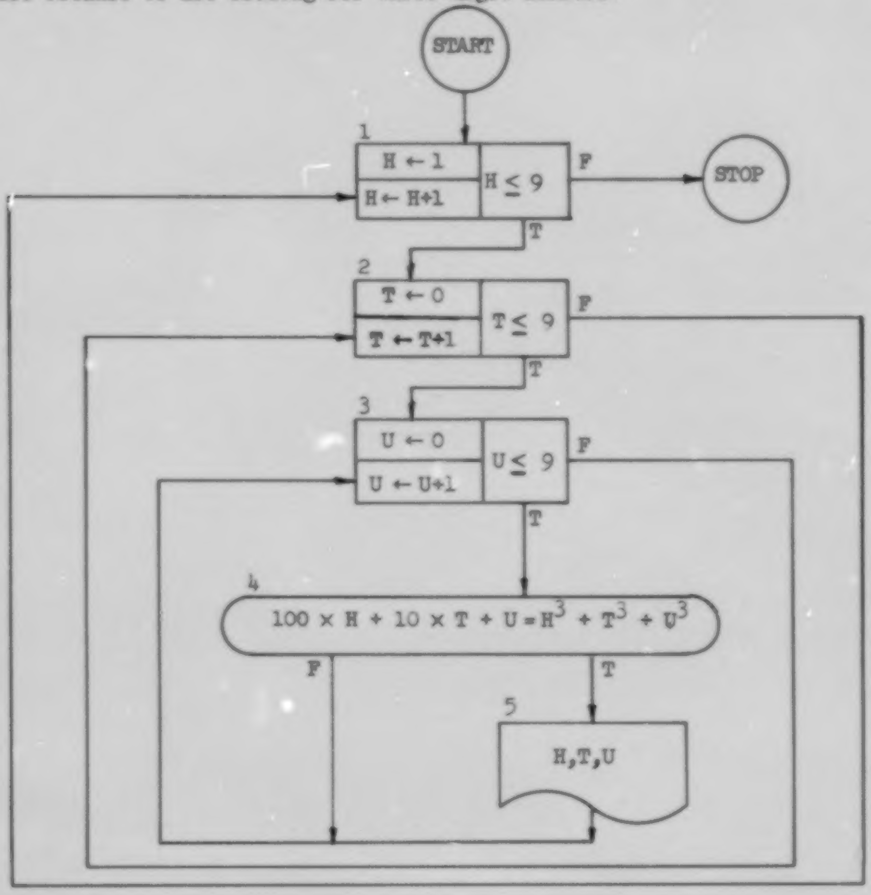

START
1
H ← 1
H ← H+1
H ≤ 9
F
STOP
T
2
T ← 0
T ← T+1
T ≤ 9
F
T
3
U ← 0
U ← U+1
U ≤ 9
F
T
4
100 × H + 10 × T + U = H³ + T³ + U³
F
T
5
H,T,U

Figure 4-31. Flow Chart for the Stickler

The stickler does not require any ingenuity--merely brute force. The 900 computations required in the algorithm would probably take all day by hand, but a fast computer would complete the calculation in less than a second.

Exercises 4-4 Set B

Re-examine Figure 4-31 and answer the following questions.

1. How many multiplications are required from START to STOP ?

2. How many different values of H^3 are computed from START to STOP ?

3. How many different values of T^3 are computed from START to STOP ?

4. Revise the algorithm so that the same value of H^3 is never recomputed and the same value of T^3 is not recomputed more than 10 times.

5. How would you revise the algorithm so that no value of H^3, T^3, or U^3 is ever computed a second time? Hint: Compute all values, 0^3, 1^3, 2^3, ..., 9^3 and store them in a separate CUBE vector having 10 elements.

6. See if you can reduce the total number of multiplications to 119 using no more than 9 different boxes in the flow chart.

7. Draw a flow chart to do the following:

   (a) Find the number of distinct (i.e., no two congruent) triangles with sides of integer length and no side greater than 100 in length.

   (b) Find the sum of the perimeters of the triangles in part (a).

   (c) In part (a) replace the condition "no side greater than 100 in length" by "with perimeter $\leq$ 100" and redraw the flow chart.

   (d) Redraw (b) with the replacement condition specified in part (c).

The Prime Factor Algorithm

In Section 4-2 (Figure 4-14) we considered the problem of finding the factors of an integer N. Now our problem is to represent N as a product of prime factors. These problems may sound similar to you. To see how they are different, compare the following. The list of factors of 360 in the order output by Figure 4-14 is

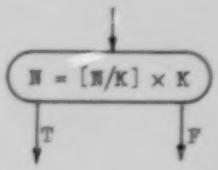
$$1, 360, 2, 180, 3, 120, 4, 90, 5, 72, 6, 60, 8,$$
$$45, 9, 40, 10, 36, 12, 30, 15, 24, 18, 20.$$

On the other hand, the complete factorization of 360 as a product of primes is

$$2 \times 2 \times 2 \times 3 \times 3 \times 5.$$

When we output the results from our algorithm, the multiplication operators will be omitted.

We will work out the algorithm following the same steps we would use doing the computation by hand. In the hand method we would check to see whether K is a divisor of N. (Start by letting K be 2.)



If K is not a divisor of N, increment K by 1 and check again. If K is a divisor of N, then:

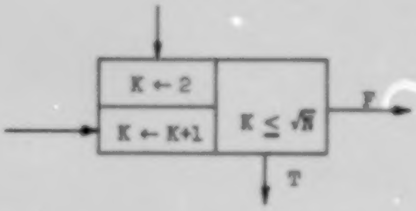
(1) print out K;

(2) replace N by N/K (so that we can now look for factors of the smaller number obtained by dividing N by the factor K);

(3) without incrementing K check whether K is a divisor of the new value of N, (remember that repeated factors are possible).

Finally, as soon as K exceeds $\sqrt{N}$, N can have no factors other than itself (and 1) so N must be prime, or equal to 1. You should satisfy yourself that, in the process we describe, the present value of N can never have factors less than the present value of K.

Since K starts at 2, is incrememted by 1, and is not to exceed $\sqrt{N}$, we evidently have the iteration box,



The rest of the algorithm has been discussed in the preceding sentences, so we exhibit its flow chart in Figure 4-32. We can see in this flow chart that as K goes up toward N, N comes down toward K. The inner loop, Boxes 3, 4, and 5, involves the check for repeated factors. The necessity for Box 6 arises from the possibility that at some point N might be a power of K. In this case successive repetitions of the inner loop would eventually reduce the value of N to 1. It is left to the student to check that nothing but primes can occur in the output. If a list of the primes less than $\sqrt{N}$ were available to be input into the computer, the computation would be considerably shortened.

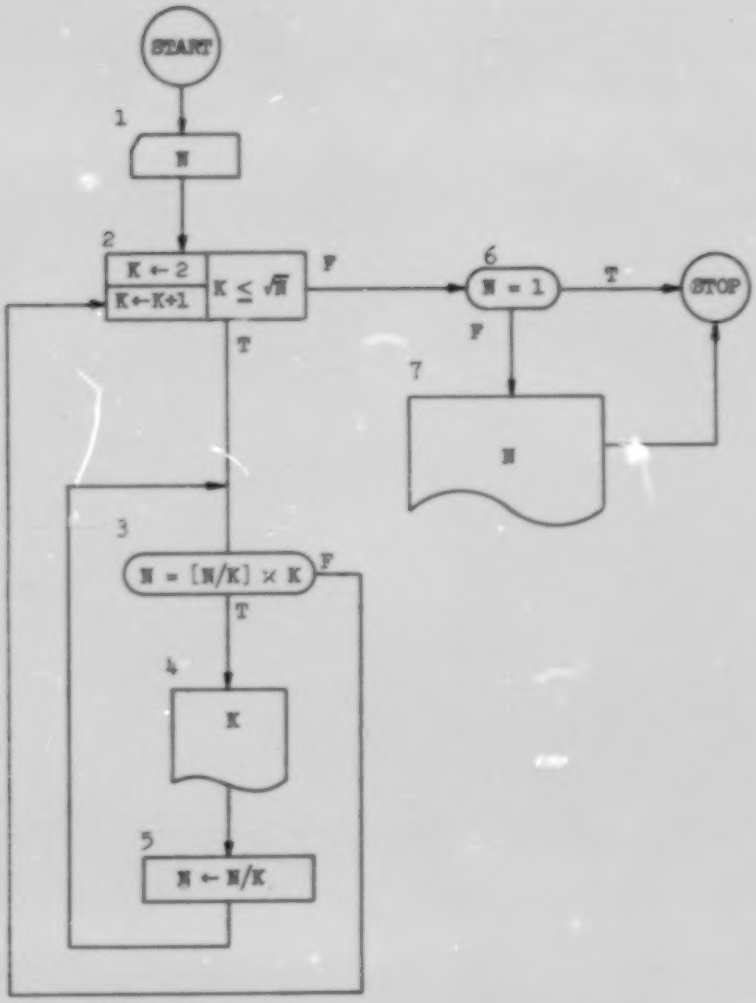

START
1
N
2
K ← 2
K ← K+1
K ≤ √N
F
T
6
N = 1
T
STOP
F
7
N
3
N = [N/K] × K
F
T
4
K
5
N ← N/K

Figure 4-32. Complete Prime Factorization Algorithm

Exercise 4-4 Set C

One of the students who studied the algorithm in Figure 4-32 wondered about ways to improve its efficiency. In particular, he was unhappy with the fact that in repeating the test in Box 2,

$$K \leq \sqrt{N},$$

we must repeatedly compute $\sqrt{N}$ even though the value of N might remain unchanged during a number of transits through the loop. As an alternative, the student developed the algorithm in Figure 4-33, claiming:

(a) it is equivalent to Figure 4-32 as far as results are concerned;

(b) while perhaps slightly more difficult to understand, it was more efficient in that $\sqrt{N}$ is computed only once for each value of N.

Your job in this exercise is to study the proposed alternative and either verify the claims made by the student, (a) and (b), or show where he is wrong. To verify or refute claim (a) you should trace through the flow chart finding the factors of several numbers like 10, 11, 12, and 24.

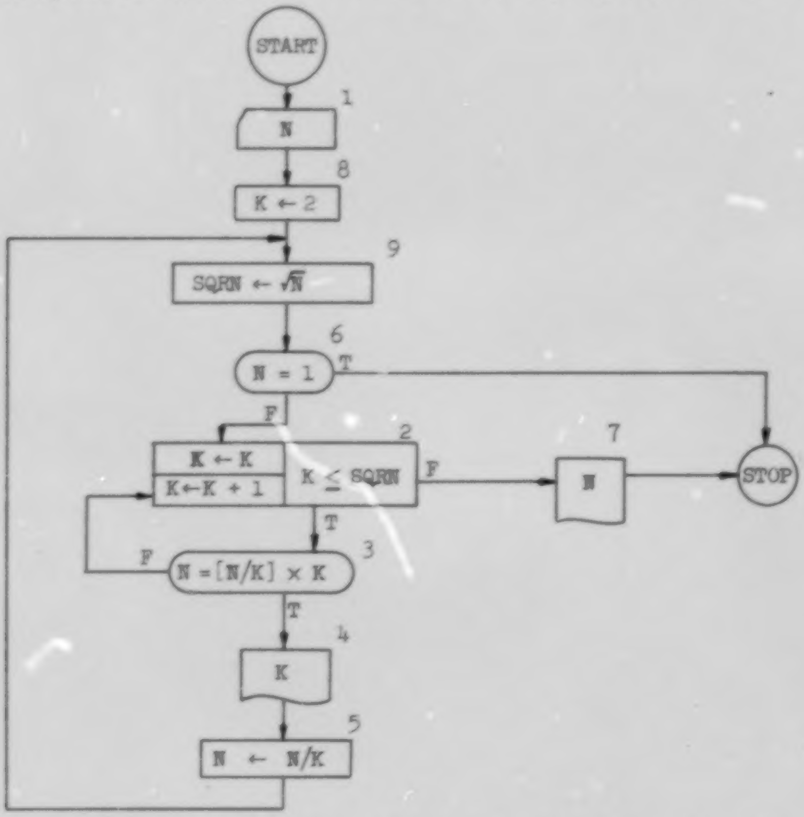
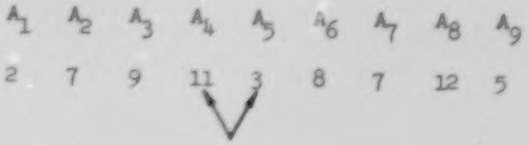


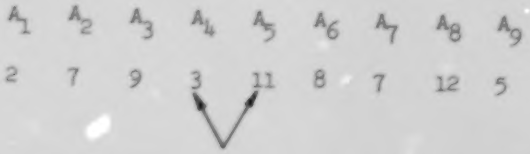
Figure 4-33. Proposed complete factorization algorithm

Shuttle-Interchange Sorting Algorithm

Look back at the sorting algorithm of Section 3-5, Figure 3-31. The purpose of the algorithm was to take a given list of numbers and "sort" or "rearrange" it in increasing order. We went through the list from left to right looking for a consecutive pair out of order.

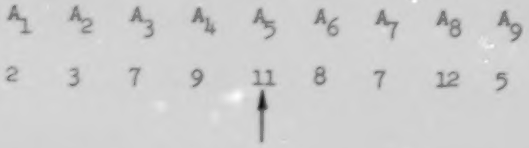


As soon as we found two adjacent numbers out of order we interchanged them.

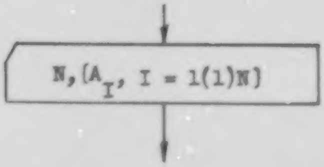


Then we started over again treating this "interchanged" list as a brand new problem. The algorithm was easy to describe but rather wasteful. The reason for this wastefulness is the rechecking of pairs preceding the interchanged pairs. These are already known to be in increasing order! We look for an algorithm to eliminate this waste.

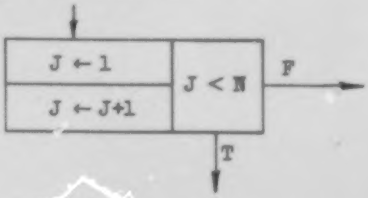
In the example above we see that the 3 is still out of place. Holding a finger on the position of 11, so as not to lose our place, we first take care of 3. Using three tests and two interchanges we work 3 back to where it belongs between 2 and 7.



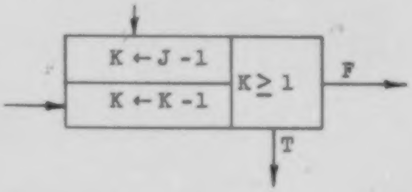
Now we come back to 11 where our finger was and compare it with the next entry and so on. Now to translate this into a flow chart, we input N and a vector with N components.



Now we introduce a variable J which runs through the subscripts picking out N - 1 pairs A_J, A_{J+1} to be compared.


$J \leftarrow 1$
$J \leftarrow J+1$
$J < N$
F
T

Then we must have a variable K which, after an interchange located by J, works the smaller of the two interchanged numbers back to its proper location.


$K \leftarrow J - 1$
$K \leftarrow K - 1$
$K \geq 1$
F
T

There are two surprises in this box. First there is a variable on the right side of the initiation compartment. Second, incrementation is negative. Both of these novelties are permissible.

Now we draw the flow chart in Figure 4-34. This sorting algorithm is quite efficient and has therefore been named. It is variously called the "shuttle-interchange" algorithm, or the "pushdown-pushup" algorithm.

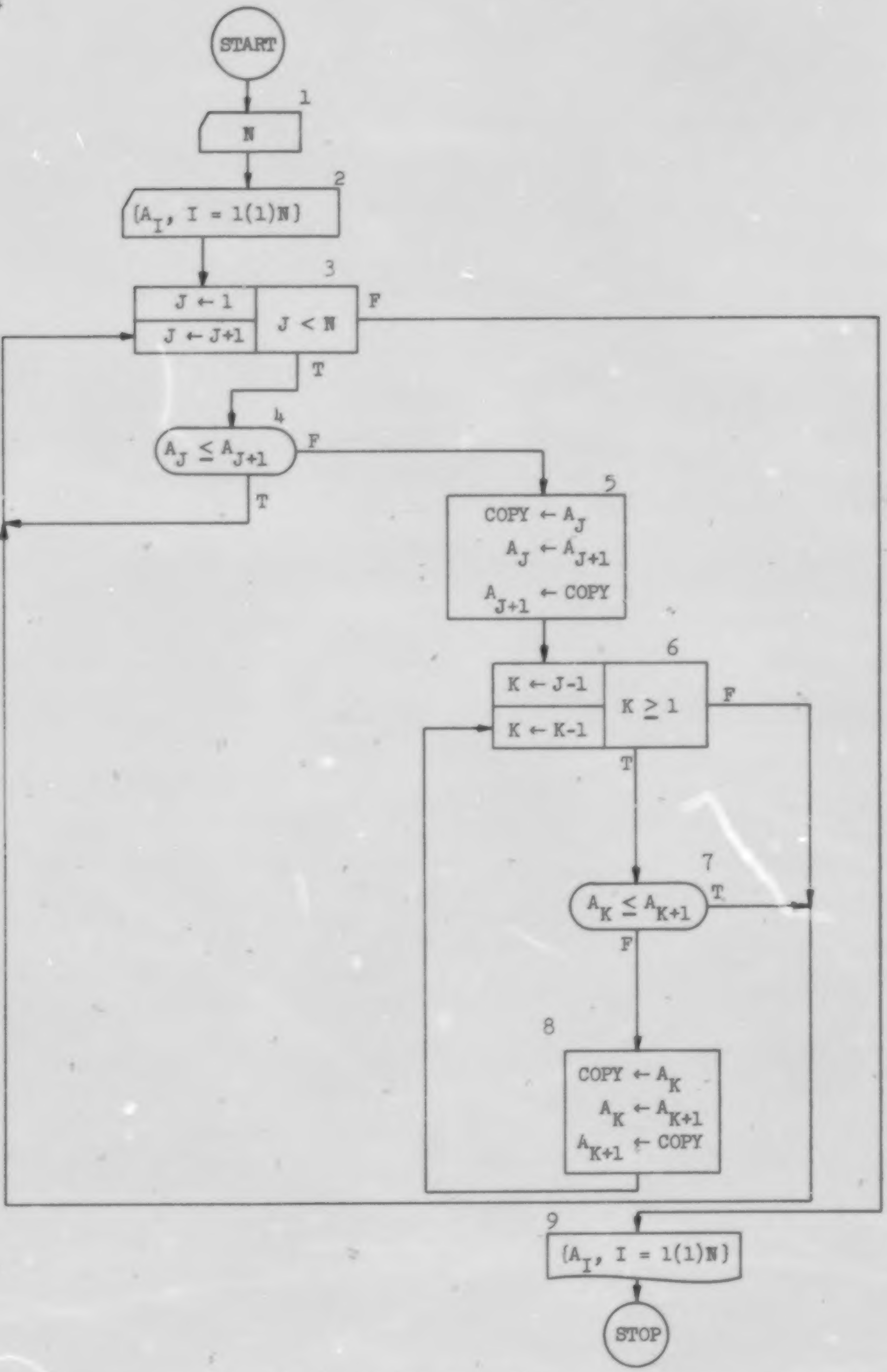

START
1
N
2
$\{A_I, I = 1(1)N\}$
3
$J \leftarrow 1$
$J \leftarrow J+1$
$J < N$
F
T
4
$A_J \leq A_{J+1}$
F
T
5
$COPY \leftarrow A_J$
$A_J \leftarrow A_{J+1}$
$A_{J+1} \leftarrow COPY$
6
$K \leftarrow J-1$
$K \leftarrow K-1$
$K \geq 1$
F
T
7
$A_K \leq A_{K+1}$
T
F
8
$COPY \leftarrow A_K$
$A_K \leftarrow A_{K+1}$
$A_{K+1} \leftarrow COPY$
9
$\{A_I, I = 1(1)N\}$
STOP

Figure 4-34. Shuttle-interchange sort

Exercises 4-4 Set D

1. In order to properly compare the primitive sorting algorithm, Figure 3-31, with the one in Figure 4-34, redraw the former using an iteration box for control of the inner loop. In redrawing Figure 3-31, you can simple omit Box 1, letting Box 8 lead to a STOP.

2. In order to appraise the efficiency of the shuttle-interchange method and to compare it with the primitive sort, we will again equate the work of sorting to the number of comparisons required. In this case, the sorting work would be proportional to the total number of times Boxes 4 and 7 (Figure 4-34) are executed.
   (a) How many times are Boxes 4 and 7 executed from START to STOP if the values to be sorted are 7, 2, -5, 4 ?
   (b) How many times are Boxes 4 and 7 executed if the values to be sorted are -9, 5, 9, 12 ?
   (c) How many times are Boxes 4 and 7 executed if the values to be sorted are 12, 9, 5, -9 ?

3. What changes would be required in the flow chart in Figure 4-34 to make it serve for sorting numbers into descending order?

4. A student brings into class the algorithm shown in Figure 4-35. He makes the following claims:
   (a) It's an algorithm for sorting numbers in ascending algebraic order.
   (b) It's more efficient than the algorithm in Figure 4-34.

   Your job is to verify or refute each claim.

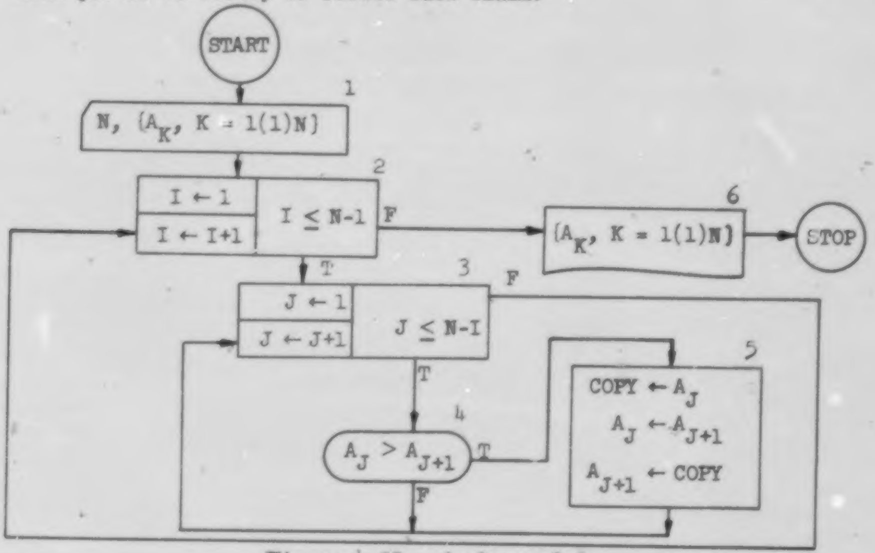


Figure 4-35. A sleeper?

[†]The Monotone Sequence Problem

A few years ago a charming little problem was making the rounds of mathematics departments. It was a true stickler in contrast with the pseudo stickler we met earlier in this section. New terms used in the statement of the problem are explained below.

> PROBLEM: Suppose you are given a sequence (that is, a list) of N numbers, guaranteed all different. Prove that the length of the longest monotone subsequence is at least $\sqrt{N}$.

By a sequence we mean a list of numbers like the components of a vector. The order in which they are written is very important. For example:

5 0 9 6 1 12 3 7 2

By a subsequence we mean the list that remains after "crossing out" some numbers in the original list. We show one of the 512 possible subsequences of the sequence exhibited above:

~~5~~ ~~0~~ 9 6 ~~1~~ 12 3 ~~7~~ 2

The reason for explaining this idea in terms of "crossing out" is to make it absolutely clear that the order of the remaining terms is not altered. By a monotone subsequence we mean one in which either the values are increasing from left to right or one in which they are decreasing.

Thus, the preceding subsequence is not monotone but the following two are monotone, the first being increasing and the second, decreasing.

~~5~~ 0 ~~9~~ ~~6~~ 1 ~~12~~ 3 7 ~~2~~
5 ~~0~~ ~~9~~ ~~6~~ ~~1~~ ~~12~~ 3 ~~7~~ 2

You can check that the increasing subsequence is the longest possible; that is to say, there is no increasing subsequence with more than 4 terms. The decreasing one is the not longest possible, since the subsequence

9 6 3 2

is longer.

In this example the longest increasing subsequence had length 4 and so did the longest decreasing subsequence. Thus, in this example the length of the longest monotone subsequence is 4.

[†] See footnote at the beginning of Section 2-8.

The problem concerning the longest monotone subsequence is actually one of proving a theorem. It may not be possible to get a computer to prove this theorem, but still this problem suggests an interesting task that a computer can perform. Namely, for a given sequence, find the length of its longest increasing subsequence.

We look for an algorithm and the first one we not only find very quickly, but also quickly reject. It begins: List all possible subsequences. Check each to see whether it is increasing. Make a note of the length of each one which is increasing. Pick out the greatest of these recorded lengths. That is a valid way of attacking the problem, and the flow chart is not difficult to draw. What, then, is wrong with this solution? We reject it because of the monstrous amount of computation. If the original sequence had 60 terms, then the number of subsequences would be 2^{60} or 1,125,899,898,650,624. For all intents and purposes, such a number of calculations may as well be infinite. An algorithm which calls for this many calculations may be of theoretical interest, but is of no practical use whatsoever.

Finding a usable algorithm for this problem is a more difficult undertaking than any we have tried so far. We won't get the idea all at once.

Let's take another look at the previous example.

I	1	2	3	4	5	6	7	8	9
A_I	5	0	9	6	1	12	3	7	2

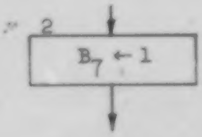
For each value of I from 1 to 9, we want to figure out the length (call it B_I) of the longest increasing subsequence having A_I as its last term. This is not difficult for the short sequence in this example. The answers are tabulated here.

I	1	2	3	4	5	6	7	8	9
A_I	5	0	9	6	1	12	3	7	2
B_I	1	1	2	2	2	3	3	4	3

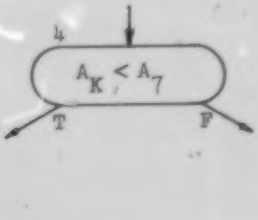
How did we find the values of B_I ? By eye--just by looking. And yet we're sure we're right. Still, there is a systematic way (an algorithm) for finding the values of the B_I. But first, note that the desired length of the longest increasing subsequence is now simply the maximum of the values of the B_I; in this case, then, it is 4.

In order to expose this systematic method, consider the preceding table only partly filled out. We will show how to find the value of B_7. We will see that the computer-inspired concept of reassignment is of great help to us in explaining this algorithm. We start by giving B_7 the initial value 1. We know that B_7 must be at least 1.

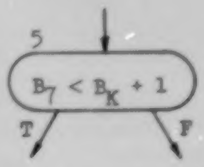
I	1	2	3	4	5	6	7	8	9
A_I	5	0	9	6	1	12	3	7	2
B_I	1	1	2	2	2	3			



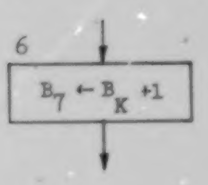
How can we find increasing subsequences ending with A_7 (i.e., ending with 3)? One way is simply to tack A_7 onto the end of a subsequence terminating with some A_K coming earlier in the list. This "tacking on the end" can only be done when it won't destroy the increasing property of the subsequence; that is, only when $A_K < A_7$. This suggests the test



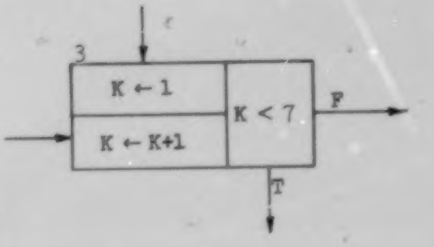
Suppose the answer is "T" (as occurs for the first time in our example when the value of K is 2). Then we know that there is an increasing subsequence whose last two terms are A_K and A_7. What is the longest such subsequence? We obtain it by finding the longest increasing subsequence terminating with A_K and then tacking A_7 on the end. The length of this "extended" subsequence will then be $B_K + 1$. To be sure we have the longest extended subsequence ending with A_7 we must compare each candidate value of $B_K + 1$ with the current value of B_7.



If true we assign the value of $B_K + 1$ to B_7.



If either of the inequalities in the decision boxes 4 or 5 is false, then no reassignment takes place. In any event we now increment K by 1 and repeat the test in Box 4. We perform this process for all values of K from 1 to 6.



All this together constitutes the Heart of our algorithm.

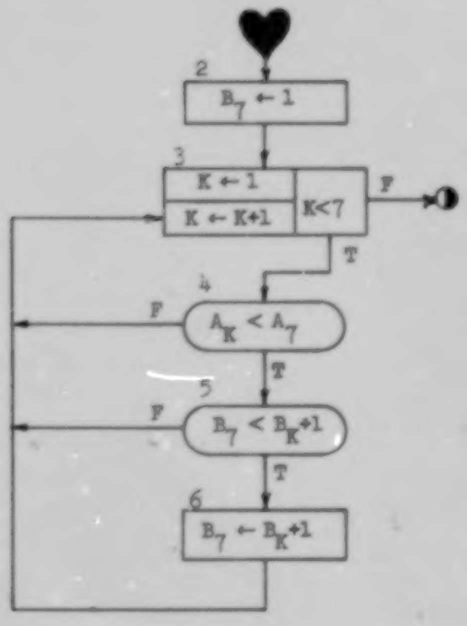
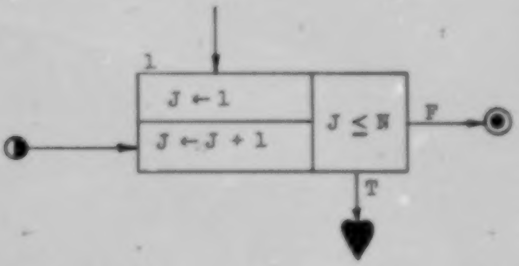

2
$B_7 \leftarrow 1$
3
$K \leftarrow 1$
$K \leftarrow K+1$
$K<7$
F
T
4
F
$A_K < A_7$
T
5
F
$B_7 < B_K+1$
T
6
$B_7 \leftarrow B_K+1$

Figure 4-36. Heart of the Algorithm

It is clear now that each of the B_J is calculated in this same way, not just B_7. In order to get this same calculation made for each B_J we replace each occurrence of 7 in Figure 4-36 by J and hang the Heart from the following iteration box,


1
$J \leftarrow 1$
$J \leftarrow J + 1$
$J \leq N$
F
T

with the connections as indicated. (We are assuming here that $(A_I, I = 1(1)N)$ has been input.)

In Figure 4-37 we see where we stand so far.

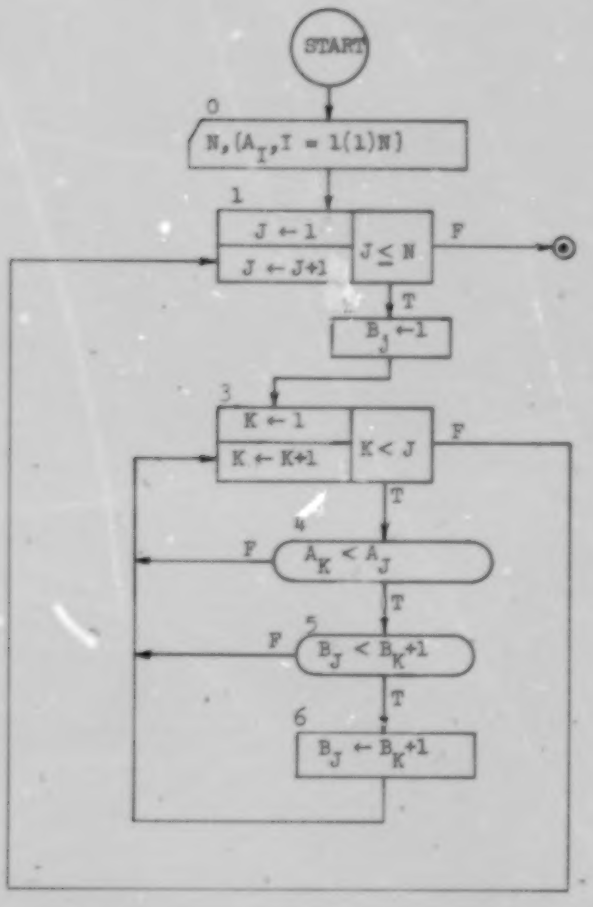

START
0
N, (A_I, I = 1(1)N)
1
J ← 1
J ← J+1
J ≤ N
F
T
2
B_J ← 1
3
K ← 1
K ← K+1
K < J
F
T
4
A_K < A_J
F
T
5
B_J < B_K+1
F
T
6
B_J ← B_K+1

Figure 4-37. Coming down the home stretch

We have done just about everything now except for getting the answer. The answer, you recall, is the largest of the values of the components of B. We have done such a computation before and it will be "child's play" to reconstruct it. The variable MAXINC is taken from MAXimum of the lengths of INCreasing subsequences.

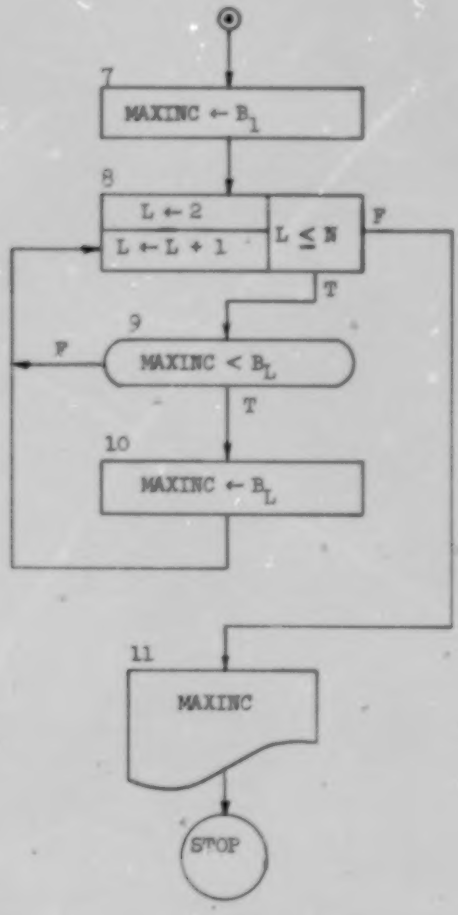


Figure 4-38. The final calculation

Now, if we join the two flow charts (Figure 4-37 and Figure 4-38) together at the bullseye, the algorithm is complete.

As usual, we ask the question, can we make any improvements? And, as usual, the answer is yes. It would produce a simpler looking flow chart as well as a slightly shorter computation to keep a "running" record of the value of MAXINC instead of introducing it after all the values of the B_J's are computed. We mean that after a given B_J is finally computed we should compare it with MAXINC. Then reassign B_J to MAXINC if B_J is larger. This eliminates one iteration box from our flow chart. The comparison and possible reassignment are seen in Boxes 13 and 14 of Figure 4-39. It is now necessary to assign an initial value to MAXINC prior to Box 1.

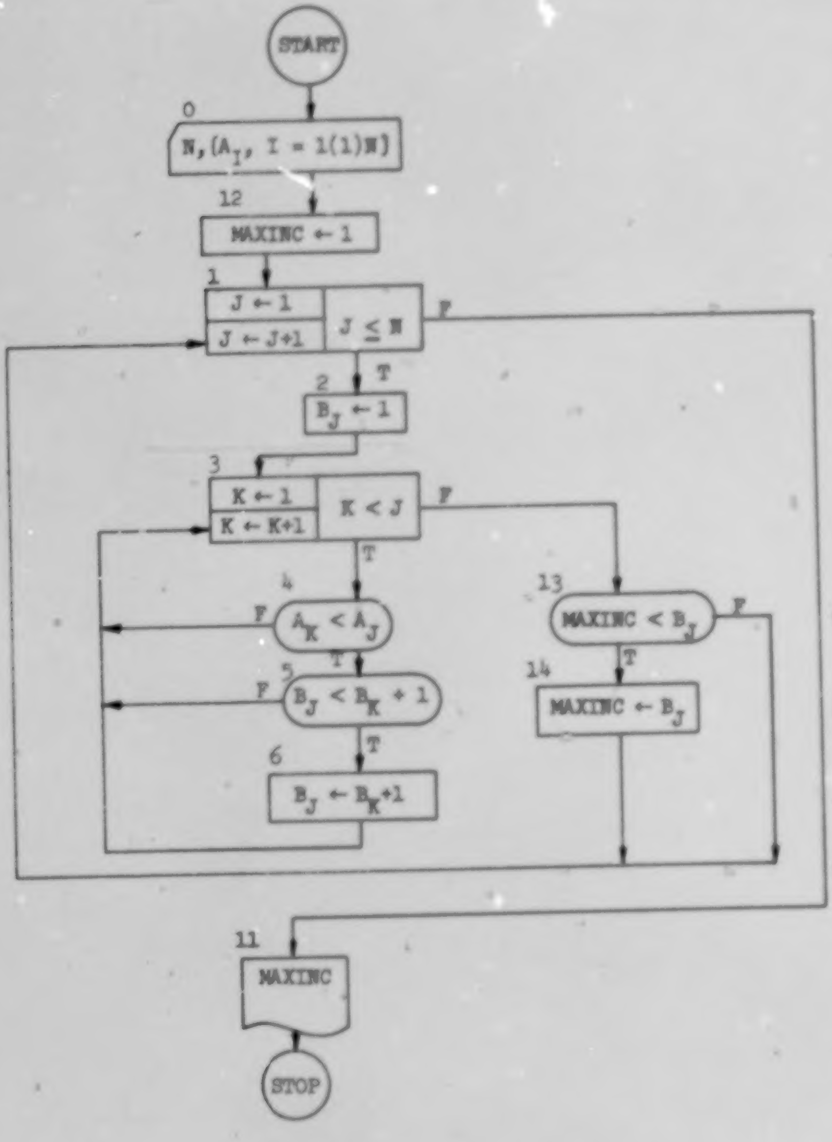

START
0
N, (A_I, I = 1(1)N)
12
MAXINC ← 1
1
J ← 1
J ← J+1
J ≤ N
F
T
2
B_J ← 1
3
K ← 1
K ← K+1
K < J
F
T
4
A_K < A_J
F
T
5
B_J < B_K + 1
F
T
6
B_J ← B_K+1
13
MAXINC < B_J
F
T
14
MAXINC ← B_J
11
MAXINC
STOP

Figure 4-39. Length of the longest increasing subsequence

As a final surprise we find that some of the machinery developed in our algorithm will inspire a proof of the theorem proposed in the original problem. This is shown in Exercises below.

Exercises 4-4 Set E

1. What changes are needed in Figure 4-39 to convert the algorithm to one which finds the length of the longest decreasing subsequence? Let C_I represent the length of the longest such sequence terminating with A_I.
2. Show that if $J < K$, then the pairs of output values

$$(B_J, C_J) \quad \text{and} \quad (B_K, C_K)$$

cannot be the same. (I.e., not both $B_J = B_K$ and $C_J = C_K$.)

3. Use the result of Problem 2 to show that the length of the longest monotone subsequence is at least $\sqrt{N}$.
4. Now that we have succeeded in producing an algorithm (we'll call it MAXY for short) which finds the length of the longest increasing subsequence, how do we find the sequence itself? One ought to be able to search through the B-vector for clues which will point to the A's belonging to this sequence. If you are on your toes, you will be able to draw a flow chart for the process of searching out and printing elements of this subsequence. The flow chart can then be tacked onto Box 11 of Figure 4-39.

To get you started we'll give you two hints:

(1) Although MAXY developed the value of MAXINC, it did not tell you where the top or "head" of the longest (or one of the longest) subsequence may be found. Your flow chart must search for it.

(2) We'll show you a picture which should suggest a plan for systematically retrieving elements of the subsequence once you have found its head. Here it is.

K	1	2	3	4	5	6	7	8	9
A_K	5	⓪	9	6	①	12	③	⑦	2
B_K	1	$\boxed{1}$	2	2	$\boxed{2}$	3	$\boxed{3}$	$\boxed{4}$	3

the head

The desired subsequence in this case is 0, 1, 3, 7.

Chapter 5

FUNCTIONS AND PROCEDURES

5-1 Reference Flow Charts

In the last few chapters you have frequently seen an assignment box like Figure 5-1.

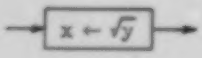


Figure 5-1. A familiar assignment box

Now we want to ask what really is meant by this box. We know that it directs us to:

1. Send an assistant to find the window box with "y" engraved on its cover and to bring the number found there to the master computer;
2. Do something with the number delivered by the assistant to produce the square root of that number;
3. Give the result to an assistant to put into the window box having "x" engraved on its cover.

How does one find the square root of a (positive) number? We know that it is not a trivial thing to do. Let's explore an algorithm for finding square roots.

Suppose we take a guess at a value of the square root of y. Call this guess g (for guess). If g is the square root of y, it is obvious that $\frac{y}{g}$ is equal to g, but we can't expect to be lucky enough to make the right guess the first time. Since the product of g and $\frac{y}{g}$ is y, we see that one of these numbers is less than $\sqrt{y}$ and the other greater.

Now suppose we make a second guess, h, at a value of the square root of y. As indicated in Figure 5-2, if the second guess, h, lies between g

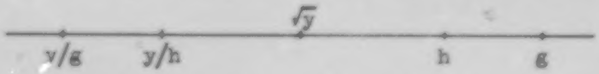


Figure 5-2. Second-guessing the square root

and $\frac{y}{g}$, so also does $\frac{y}{h}$. So, if we take any point h in the interval with endpoints g and $\frac{y}{g}$, we can define a new interval with endpoints h and $\frac{y}{h}$ contained in the first interval and still containing $\sqrt{y}$. If we make repeated guesses by taking a new point inside each new interval, the intervals must get smaller and smaller without end and the guesses must come closer and closer to $\sqrt{y}$.

What point in this interval with endpoints g and $\frac{y}{g}$ would you like to take for a second guess? Any one will do but some might do better than others. An easy point to find (and an excellent one) is the halfway point,

$$h = \frac{1}{2}(g + \frac{y}{g}) .$$

If we continue to make our successive choices in this way, how fast will the successive intervals get small?

Let d_1 be the length of the first interval,

$$d_1 = |g - \frac{y}{g}| = \left|\frac{g^2 - y}{g}\right| ,$$

and let d_2 be the length of the second interval,

$$d_2 = |h - \frac{y}{h}| = \left|\frac{g^2 + y}{2g} - \frac{2yg}{g^2 + y}\right| .$$

Now you can easily check that

$$d_2 = \left|\frac{(g^2 - y)^2}{2g(g^2 + y)}\right| = \frac{1}{2}\left|\frac{g^2 - y}{g^2 + y}\right| \times d_1 .$$

Since y is a positive number,

$$\left|\frac{g^2 - y}{g^2 + y}\right| < 1$$

so that

$$|d_2| < \frac{1}{2}|d_1|.$$

Therefore, each new interval must be less than half as long as the last one. In fact, after only a few of these successive choices the new intervals will be much less than half as long as their predecessors.

Thus, the process of repeatedly averaging g and $\frac{y}{g}$ and assigning the result to g is guaranteed to yield successively better approximations to the square root of y --even if the first guess is really terrible.

We draw a flow chart of this process, Figure 5-3.

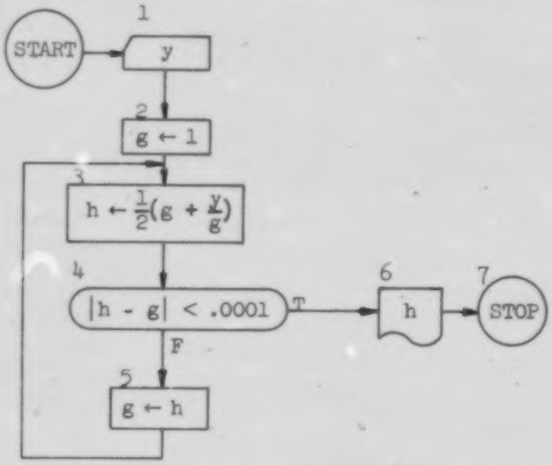


Figure 5-3. Square root method (due to Newton)

Box 4 shows a condition we could use to end this iterative process when the improvement in the approximation becomes less in magnitude than some number like .0001.

In case we ever want to take a square root again, wouldn't it be a good idea to file a permanent copy of Figure 5-3 in a notebook so everyone could use it? Besides, suppose we ever need to take square roots at more than one place in a flow chart, wouldn't it be handy to have a reference flow chart that could be referred to from any place in another flow chart?

To help us make a permanent reference copy of the square root flow chart, we will adopt a few new conventions. We need another way to indicate the argument of the square root, other than by reading a card as in Figure 5-3.

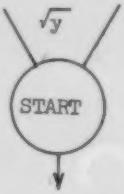


Figure 5-4. A funnel

A new flow chart shape is used to show both the purpose (taking a square root) of this reference flow chart and the argument, y. This new shape is the funnel shown in Figure 5-4. We also need a way, in the flow chart language, to state the result and to say that we now return to that box in the flow chart that caused the reference. This shape, replacing the print and stop boxes of Figure 5-3, is the "return" box (Figure 5-5).

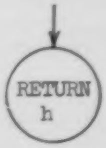
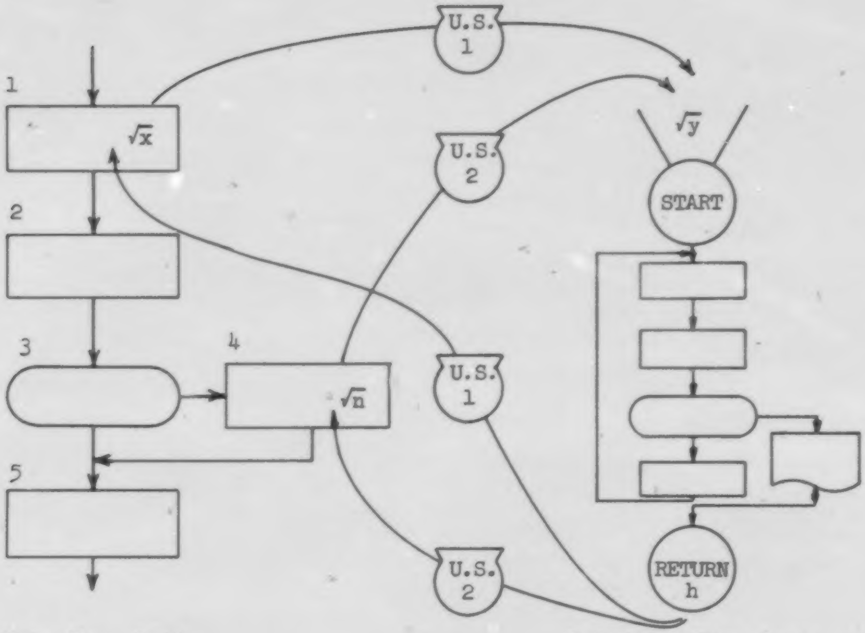


Figure 5-5. Return box



Main flow chart

Reference flow chart

Fig. 5-6. Use of a reference flow chart

Figure 5-6 shows the use of the funnel and the return box on the ends of a reference flow chart. The first time the reference flow chart is required (by $\sqrt{x}$ in what we will call the "main" flow chart), we go to the funnel end via route 1. The inscription in the funnel says that, to use the reference flow chart, we must first assign the value of x to y so that whenever y appears in the reference flow chart its initial value will be the value of x. When execution of the reference flow chart is finished we are to return via route 1 to the same box in the main flow chart that caused the reference and complete the execution of this box. When we again require the reference flow chart (for $\sqrt{n}$, in box 4) we are to go to the funnel via route 2, assign the value of n to y, execute the reference flow chart, and return to the main flow chart via route 2. A flow chart for $\sqrt{y}$ can be referred to as often as necessary by a main flow chart.

Figure 5-7 shows a reference flow chart for taking square roots.

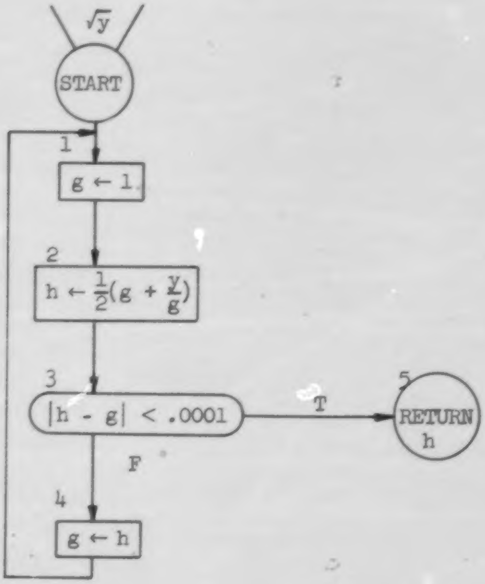


Figure 5-7. Reference flow chart for square roots

Exercises 5-1

1. For a cube root the iteration formula could be

$$h \leftarrow \frac{1}{3}\left(2g + \frac{y}{g^2}\right).$$

Prepare a reference flow chart for the calculation of a cube root.[†]

2. Prepare a reference flow chart for the evaluation of $f(x) = 3x^2 - 2x + 1$.

3. Prepare a reference flow chart for ABSOL(X) which evaluates the absolute value of x.

[†] Since $g^2 \times \frac{y}{g^2} = y$, it is clear that g and $\frac{y}{g^2}$ lie on opposite sides of $\sqrt[3]{y}$. It is therefore natural to choose the average, $\frac{1}{2}(g + \frac{y}{g^2})$ for the next guess. In later math courses it is shown that the weighted average, $\frac{1}{3}(2g + \frac{y}{g^2})$ is the better choice.

†5-2 Mathematical Functions

Many of you have been introduced to the mathematical concept of functions in earlier courses. Even if you have not, you have all dealt with particular functions such as the trigonometric, inverse trigonometric, the logarithm and exponential functions, and the square root function. In computing, the word "function" is used in a subtly different way than elsewhere. To enable you to appreciate the difference, we will discuss here the mathematical concept of function in detail. We have touched on the function concept very casually in earlier chapters of this book.

The first basic idea involved in the function concept is "unambiguous designation." Suppose we make a remark about Elinor's hat. It turns out that Elinor has a whole closet full of hats and it is not at all clear which one we are referring to. But now a remark is made about Elinor's head. Ah! That is a different matter. Elinor has but one head, so that the head being referred to is perfectly clear, that is to say, unambiguously designated (or determined).

We think of a function as being a thing which performs such unambiguous designations. But a function makes not just one unambiguous designation, but a whole lot of unambiguous designations. For example, if we were to say "the hat on the head of X", we would have an unambiguous designation when a particular person's name is inserted for X, and a large set of hats could be designated depending on the set of names we allow to be used for X.

This is the second basic idea in the function concept. There is a set called the domain of the function and for each member of this set the function unambiguously designates something.

As an example consider a function which we will call BRTHDY. The domain of this function is the set of all human beings. Whenever this function is presented with a member of its domain, it designates the birthday of that person. If the function is presented with Abraham Lincoln it designates February 12. We write

$$\text{BRTHDY(Abraham Lincoln)} = \text{Feb. 12}$$

to indicate that Feb. 12 is the thing designated by the function BRTHDY on being presented with Abraham Lincoln. Similarly,

$$\text{BRTHDY(George Washington)} = \text{Feb. 22}\ .$$

[†] This section can be omitted if you are already familiar with the mathematical concept of a function.

What is the value of

BRTHDY(Tom Spumoni) ?

We don't know. But we do know that Tom Spumoni has a birthday (provided he is a real person) and it is this birthday which is designated by the expression

BRTHDY(Tom Spumoni).

(In mathematics, as distinguished from computing, we are satisfied here with the existence of Tom Spumoni's birthday and we do not feel the need of being able to exhibit it explicitly.)

Another example of a function is the squaring function, SQR. This function, given any real number, designates the square of that number. Thus,

SQR(2) = 4, SQR(-3) = 9,

SQR(1.7) = 2.89, SQR(0) = 0,

SQR(3) = 9

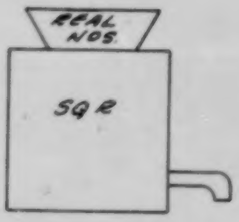
So now we see what a function does. If, to each member X of a given set, a function called F is applied, then an object, F(X), is unambiguously designated. That tells you what a function does, but you may well want to know what a function is. This, it turns out, is not terribly important to us.

The situation here is somewhat similar to that of numbers. We know how numbers behave under various operations; we know numerous properties of the set of numbers such as order and density; we know properties of various subsets such as the integers and the rationals. In short, we know almost everything about numbers--except what they are. And we have been able to operate with numbers quite adequately for lo these many years without this knowledge.

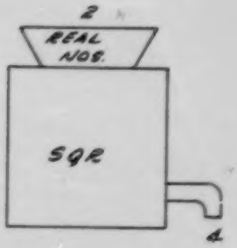
And so it is with functions. As long as we know that a function, F, will, for each member, S, of its domain, produce for us the "functional value", F(X), we have no need in mathematics of knowing how this is done. There is an analogy with dialing a telephone number. We know that for each telephone number in the "domain" (i.e., the set of phone numbers currently in service) a certain telephone is unambiguously determined. When we dial the number, this telephone rings. We are not concerned with the wiring, the relays, the switches, the computers, and the coaxial cables which may be involved in the process of making the phone ring at the other end of the line. It is only important to us that this phone does ring.

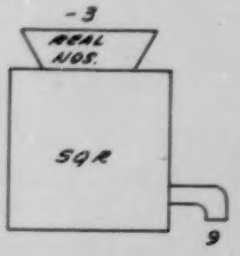
Now that we have been assured that it is not important for us to know what functions are, we will discuss three ways of "representing" them or thinking about them. This is similar in spirit to our way of representing the real numbers as points on the number line. This means, more or less, that we think of the real numbers as being points on a line.

The representation of a function as a machine is useful as well as popular. Consider, for example, the squaring machine as a representation of the function SQR.



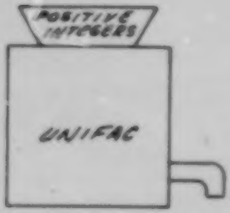
We see that the machine is equipped with an input funnel and an output spigot. We have indicated the domain of the function (the permissible input) on the input funnel. If we input 2, the machine grinds and cranks and outputs the number 4.



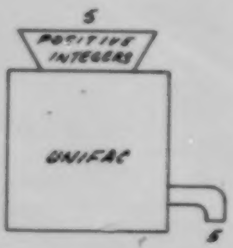


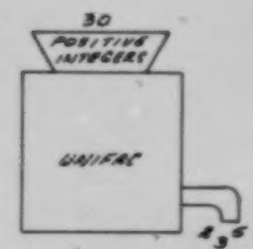
If we input -3 the machine whirrs and clanks and outputs 9. In general, for any value of X that we input, the machine outputs X^2. The important thing, the thing that makes SQR a function, is that for each input value there is just one output value.

Here is another machine, called UNIFAC for UNIque FACtorization.

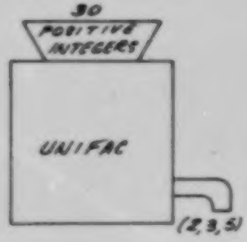

POSITIVE INTEGERS
UNIFAC

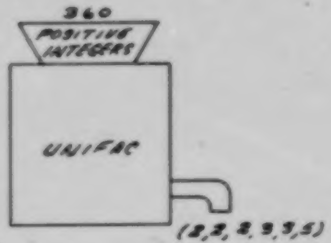
The domain of this function, as we see, is to be the set of positive integers. If we input an integer, the output is the unique decomposition of that integer into its prime factors. So we input 5 and since 5 is prime, 5 is output.

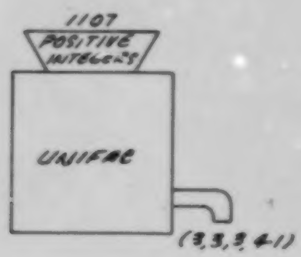

5
POSITIVE INTEGERS
UNIFAC
5


30
POSITIVE INTEGERS
UNIFAC
2 3 5

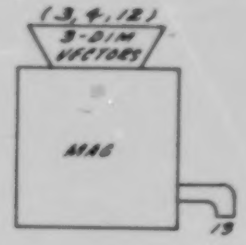
That is, UNIFAC(5) = 5. Suppose we input the integer 30. Its prime factorization is 2 × 3 × 5, so what is the value of UNIFAC(30) ? Is it 2 ? or 3 ? or 5 ? or do we just take our choice? An ambiguous answer here would mean that UNIFAC would not be a function. We build the machine so as to output, not a cascade of factors, but a single vector, having as components the factors arranged in non-decreasing order.


30
POSITIVE INTEGERS
UNIFAC
(2,3,5)


360
POSITIVE INTEGERS
UNIFAC
(2,2,2,3,3,5)


1107
POSITIVE INTEGERS
UNIFAC
(3,3,3,41)

An example of a function which accepts vectors as input is the function MAG which accepts three-element vectors as input and outputs their magnitudes (i.e., the square root of the sum of the squares of the components).


(3,4,12)
3-DIM VECTORS
MAG
13

Now with SQR, UNIFAC, and MAG we have illustrated functions which, for an element of the domain (whether its value is a single number or a vector), unambiguously designate an element of the range (either a single number or a vector). In each case we haven't cared about how this designation happened. However, the fellow who has to design the telephone system, or the UNIFAC machine, or a reference flow chart, must be able to spell out, as a rule or recipe, a way to find a value of F(X) once a value of X is given (Figure 5-8). This is another way of thinking of a function.

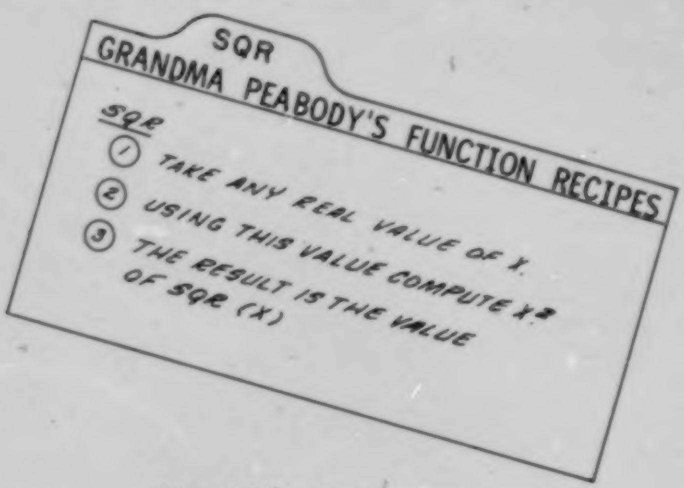

SQR
GRANDMA PEABODY'S FUNCTION RECIPES
SQR
① TAKE ANY REAL VALUE OF X.
② USING THIS VALUE COMPUTE X²
③ THE RESULT IS THE VALUE OF SQR (X)

Figure 5-8. Spelling out a recipe

When specifying a function we must be careful to indicate the domain of the function. If no such indication is made, then it will be assumed that the domain is the set of all things for which the rule makes sense. Here are three examples.

Example 1: $F(x) = 3x^2 + 5x - 2$ for $-3 \leq x < 5$.

Example 2: $G(x) = \dfrac{x^2 - 1}{x^2 + 1}$

Example 3: $H(x) = \sqrt{1 - x^2}$

In the first of these examples we see that the domain is specified to be the half-open interval $[-3, 5]$. In Example 2 the domain is not specified and so it is assumed to be the set of all real numbers since $(x^2 - 1)/(x^2 + 1)$ is meaningful for all real values of x. In the third example the domain is again unspecified but this time the domain is $[-1, 1]$ since $\sqrt{1 - x^2}$ is not meaningful for values of x taken from outside this interval.

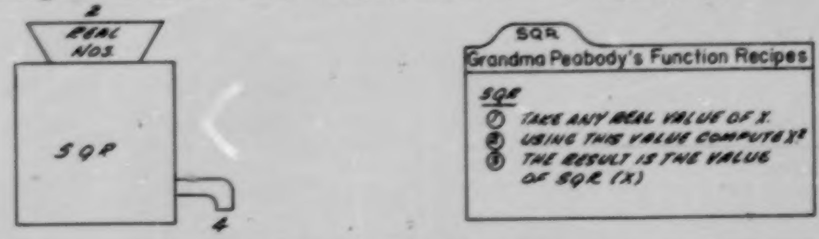
One point which should be clarified at once is: Such a rule does not depend on the variables used in expressing it. For example, if we write

$$J(t) = \frac{t^2 - 1}{t^2 + 1}$$

then the function J is identical with the function G of Example 2.

We see that this "rule" viewpoint strongly suggests that of the programmer who is to draw a reference flow chart. The reference flow chart, in Section 5-1, for finding square roots is an algorithm and an algorithm is, after all, a kind of rule or recipe. On the other hand the "machine" viewpoint suggests more nearly the view that might be adopted by the programmer drawing a main flow chart who wants to use the reference flow chart. All he cares about is that if he provides a value of x to the reference flow chart for the square root, it will return a value he can use as the square root of x.

Whether the "machine" viewpoint or the "rule" viewpoint is adopted in thinking of functions we can tabulate results as in Figure 5-9.


2
REAL NOS
SQR
4
SQR
Grandma Peabody's Function Recipes
SQR
① TAKE ANY REAL VALUE OF X.
② USING THIS VALUE COMPUTE X²
③ THE RESULT IS THE VALUE OF SQR (X)

X (input)	-10	-3	-1.7	-1	0	1	1.4	2	1.7	5
F(X) (output)	100	9	2.89	1	0	1	1.96	4	2.89	25

Figure 5-9. Tabulation of SQR

The table given in Figure 5-9 is very closely akin to the set of ordered pairs: (-10,100), (-3,9), (-1.7,2.89), (-1,1), (0,0), (1,1), (1.4,1.96), (2,4), (1.7,2.89), (5,25). (We call them "ordered" pairs to make it clear, for example, that (3,4) is not to be considered the same as (4,3). I.e., the order is taken into account.) The first entry in an ordered pair is called the "abscissa" and the second is called the ordinate.

$$(\overset{\underline{\text{abscissa}}\text{ (input)}}{-1.7},\overset{\underline{\text{ordinate}}\text{ (output)}}{2.89})$$

Although we cannot write out an infinite table, we can conceive of an infinite set of ordered pairs. There is a mathematical notation to describe it. The mathematical notation describing the set of ordered pairs which accompanies the SQR function is

$$\{(x,y)|y = x^2\}.$$

When you read this out loud you say, "The set of all ordered pairs of the form (x,y) which satisfy the condition that y is equal to x squared."

The rule viewpoint and the set of ordered pairs viewpoint are really equivalent. For, firstly, the rule determines the set of ordered pairs. And secondly, the set of ordered pairs itself can be thought of as a rule equivalent to the given one. We explain what this means immediately.

Suppose someone asks what the value of SQR(2) is. We could tell him to, "go to that set of ordered pairs over there and hunt for the pair whose abscissa is 2. The ordinate of that pair is SQR(2)." That is a rule for finding SQR(X). And this rule is equivalent to the original rule because the two rules always give the same results.

Many people like to think of the function as actually being the set of ordered pairs. They then accept the following definition: "A function is a set of ordered pairs with the property that no two pairs in the set have the same abscissa." (The qualification about the abscissa expresses the unambiguous designation property.)

This "set of ordered pairs" viewpoint is evidently closely akin to that adopted in Chapter 4 in the discussion of table-look-up, especially in the case that the table is punched on a stack of cards. In that case, each card may be thought of as an ordered pair.

Whatever viewpoint we adopt, there is one essential point we must be agreed on. That is, if we have two functions, F and G, having the same domain, A, and such that

$$F(X) = G(X) \text{ for all } X \text{ in } A,$$

then F and G are in fact the same function.

We must be especially careful to be clear on this point if we adopt the "rule" viewpoint. For two statements of rules can look quite different but in fact be equivalent. Then if we give the rule for finding Q(X) as: Take the numbers one more and one less than X and add 1 to their product,

$$Q(X) = (X + 1) \times (X - 1) + 1$$

then we find that Q is the same function as SQR.

Mathematically, every unambiguous designation is a function so that every computer program can be viewed as a function. The common computing usage of the word function refers to the use of a separate subprogram for the computation of values. Simple functions for which the arithmetic operations of addition, subtraction, multiplication and division are sufficient are usually computed in the main program (unless they occur so often that a subprogram becomes more economical of memory space). Such simple functions (e.g., $x^2 + 1$) are, of course, recognized as being functions but are seldom called by that name.

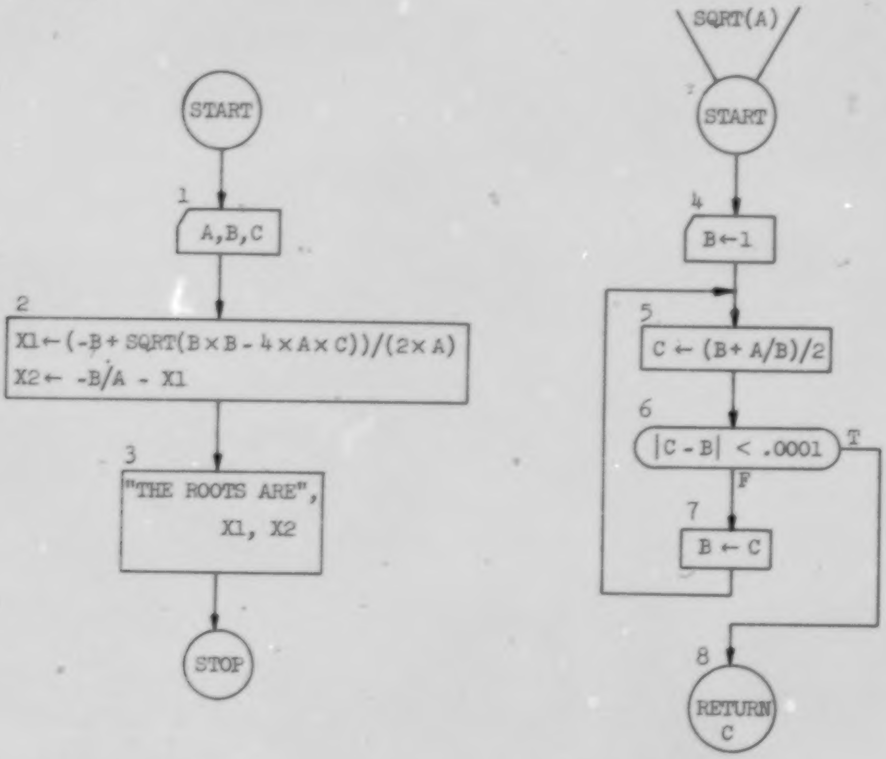
Finally, in computing we will not be interested in functions for which functional values cannot in principle be computed even when the existence of functional values can be proved. The class of functions we will be interested in are called "computable functions".

Since computer calculations on the reals are almost always approximate, you should recognize that, in effect, we replace mathematical functions by computer functions which are often slightly different.

5-3 Getting In and Out of a Functional Reference

We are now ready to tie up the ideas developed in the first two sections of this chapter. In Figure 5-7 we saw an elegant algorithm for computing approximate values of the square root function. We have explained that this algorithm may be "referred to" by another algorithm and we want to see just how. To streamline our terminology we shall frequently substitute the word "subroutine" for "reference flow chart".

First of all we will bring to light a source of confusion so as to properly exterminate it. In Figure 5-10 we see two flow charts. On the left is the main flow chart and on the right is a subroutine. The program on the left is immediately recognizable as the computation of roots of the quadratic equation, $A \times X^2 + B \times X + C = 0$ by the quadratic formula (where we never input 0 for A). The subroutine on the right is the same as that in Figure 5-7 except that the variables have been changed so as to create the necessary confusion.



(a) Main flow chart

(b) Functional reference flow chart

Figure 5-10. A source of constructive confusion

We first analyze the connection between the two flow charts of Figure 5-10 in a very naive manner. Suppose for this discussion that the input values of A, B, C are 1, -4, 3. According to the principles of Section 2-5 the first part of the expression on the top line of box 2 is $B \times B - 4 \times A \times C$. This evaluates to 4. Now the evaluation process calls for SQRT(4) to be evaluated. As we would expect, this means that we drop 4 into the funnel of the reference flow chart. This is equivalent to assigning 4 to the variable A appearing in the funnel. The wheels are thus set in motion in the subroutine. During execution of the subroutine, values are assigned to B and C each time through the loop. The result seems to be that although on the one hand the desired square root is correctly evaluated, the values of the variables A, B, and C are altered in the process. This was clearly not anticipated by the person who drew the flow chart for the main program. He innocently called for the evaluation of a certain square root with no idea that the values of his precious variables would be altered in the process.[†]

The first method which comes to mind for handling this dilemma is to use entirely different variables in Figure 5-10b than those used in Figure 5-10a. This is equivalent to using the flow chart of Figure 5-7 in place of Figure 5-10b. It is true that if we adopt this policy we will be in no danger of confusion in reading the flow charts and if we trace them out by hand we will compute good approximations of the roots. (In fact, in the example of the previous footnote we would, by a fortuitous fluke of the rounding method chosen, obtain the exact values of the roots.)

If we accept the solution offered in the last paragraph, then we will lose a golden opportunity to comprehend what really goes on in computer subroutines. You must be wondering by now what the right answer is to the dilemma that confronts us. Since we have earlier taken the viewpoint that the variables are the symbols themselves, we do not want to claim that the symbols A, B and C of Figure 5-10(b) are different from the symbols A, B and C of Figure 5-10(a). Yet this does suggest the right idea. In order to get hold of this idea we fall back once more on our ever-dependable window boxes.

[†] You might be interested in following this "naive approach" by hand through the flow charts of Figure 5-10. Using input values of 1, -4, 3 and using 4 tabular digit round off with **TRUNK** (See Section 2-5), the values of A, B, C will be 4, 2.05, 2 when box 8 is reached. Box 8 tells us that the present value of C is the desired value of the SQRT(4). Continuing the computation with what appear to be the present values of A, B and C, we obtain -.00625 and -.5062 as output values of X1 and X2.

We will have to make our window box model of computing a little more elaborate in order to handle subroutines. We are going to think of our subroutine as being carried out inside a sealed brick chamber (see Figure 5-11). The only contacts this chamber has with the outside world are a funnel on the top and a window in the side. The number to be square-rooted is put into the funnel whereupon the subroutine is executed and the desired square root appears in the window.

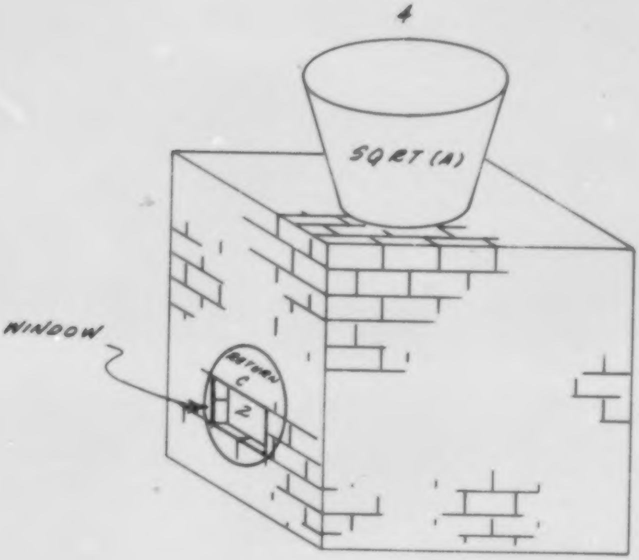


Figure 5-11. Square root subroutine (exterior view)

Inside the sealed chamber there is a separate staff (master computer, assigner and reader) and window boxes for each of the variables appearing in the subroutine. These boxes are completely inaccessible from the main flow chart with two exceptions. We can assign to the variable appearing in the funnel when a square root is called for in the main flow chart. And we can read from the variable appearing in the RETURN circle.

If some or all of the variables appearing in the subroutine also appear in the main flow chart, as in Figure 5-10, then there will be additional window boxes for these variables outside the sealed chamber. All references in the main flow chart are to boxes on the outside. All references in the subroutine are to boxes on the inside.

In Figure 5-12 we show an interior view of the square root flow chart of Figure 5-10b but with the personnel not included.

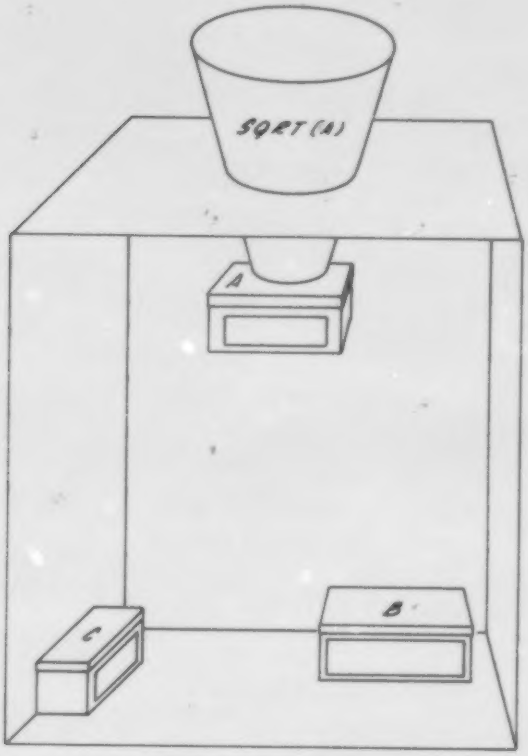
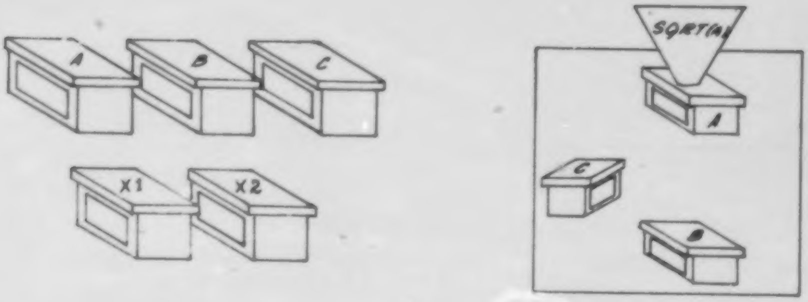


Figure 5-12. Square root subroutine (interior view)

There is a slight modification in one of the window boxes. The window box labeled C has windows on both sides. One of them is right up against the outside window. In this way this box can be read both inside and outside the room. When we assign a value to the variable A through the funnel, this starts the subroutine in motion and there will be no further contact with the outside world until the subroutine is completed.

Next we show in Figure 5-13 all the window boxes used in the main flow chart and the subroutine of Figure 5-10.



a) Main

b) Subroutine

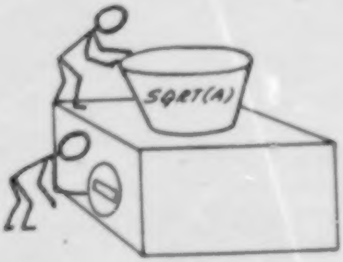
Figure 5-13. Window boxes for Figure 5-10

Now you should see that nothing here in the subroutine will change the values in the window boxes of the main flow chart.

With the subroutine enclosed in these brick walls it is almost as though the two flow charts were in different worlds. From the subroutine we cannot assign to variables of the main flow chart or even see what these variables are. And from the main flow chart we similarly have no contact with the variables of the subroutine except for two points of contact. Namely, that we can assign to the variable in the funnel and read the value of the one in the return box, but even in these cases we pay no attention to what symbols are used, but only the location in which they occur. From the point of view of the main flow chart, the variables in the funnel and the return box merely play the role of "place holders."

Concerning the variables in the subroutine it is sometimes useful to think of them as being of two kinds. Those which do not appear in the funnel and hence cannot be assigned to directly from the outside are called "local" variables. Those which do appear in the funnel provide the only link with the outside world. They are called non-local variables.

We are now prepared to finish the description of the interplay between the main flow chart and the subroutine. Again we illustrate with the pair of flow charts of Figure 5-10. When in box 2 of this figure the value of SQRT (B × B - 4 × A × C) is called for, the master computer sends out the assigner and the reader as a pair. They hunt for the chamber with SQRT on the funnel. (They pay no attention to the variable that follows SQRT, because the less they know about the variables inside the chamber the better.) When they find this chamber, the assigner goes to the funnel and the reader goes to the window. The assigner drops the value of $B \times B - 4 \times A \times C$ into the funnel (thus assigning this value to the variable A of the subroutine). When the wheels stop turning inside the chamber, the reader reads the value through the window and returns it to the master computer to use in the computation. (To those who have read Appendix A we explain that in SAMOS this value goes into the Accumulator.)[†]



It is important to note that the value of this square root does not get assigned to any of the window boxes of the main flow chart (at least not directly). The master computer in performing his arithmetic computations can only receive one numerical value at a time. For this reason there will always be only one window on the functional reference sealed chamber. Hence, the functions with which we are dealing will always have for their functional values a single numerical value (never a vector). In present day mathematical terminology such functions are called "functionals." Vector-valued functions are also very important in computing work but they will always be evaluated by the use of techniques for "procedures," to be taken up in the next section.

Functions of more than one variable (i.e., functions whose domains are sets of vectors) are treated in almost the same way. Recall the function discussed in Chapter 3 for finding the minimum of the values of B and C. The rule expressing this function is

$$\mathrm{MIN}(B,C) = \begin{cases} B, & \text{if } B \leq C \\ C, & \text{if } B > C. \end{cases}$$

The reference flow chart for this function as given in Figure 5-14 clearly follows this rule.

[†] Some computers do not have registers called "accumulators". Nevertheless, all computer systems designate some component to hold the result of a subroutine.

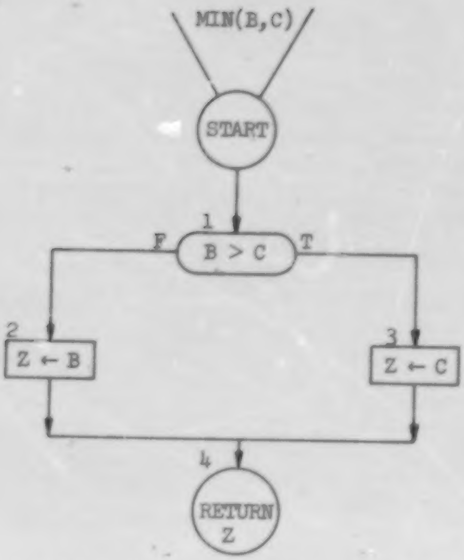


Figure 5-14. Reference Flow Chart for MIN(B,C)

When a main flow chart needs to refer to Figure 5-14, it simply uses the name MIN together with the arguments whose values are to be "put in the funnel."

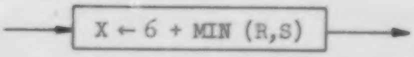


Figure 5-15. Reference to the MIN flow chart

Figure 5-15 tells us that we should find a subroutine called MIN, assign the values of R and S, respectively, to the reference flow chart variables appearing in the funnel and then execute the reference flow chart. Then we are to read the value of the variable in the return box and return with it to the main flow chart. Here we add 6 to this value and assign the result to Y.

Again we think of the subroutine as being carried out in a sealed brick chamber.

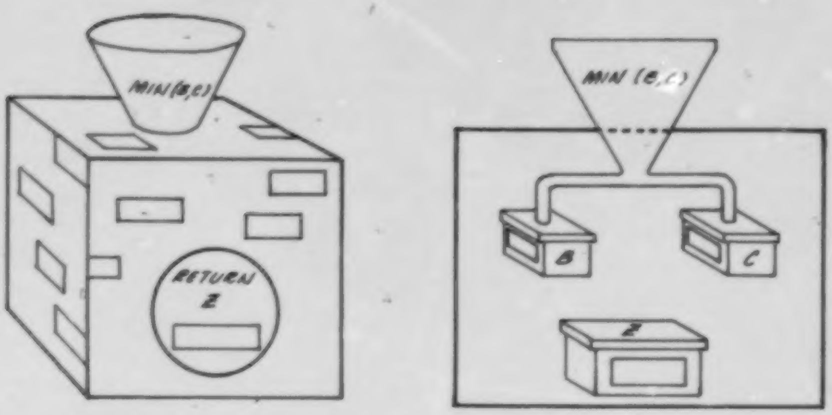


Figure 5-15. Sealed chamber for MIN(B,C), exterior and interior views

Here if the main flow chart calls for the value of MIN(R,S), then the values of R and S are dropped in the funnel in that order and are assigned in that order to the subroutine window boxes of B and C. Again we are not to be confused by the possible occurrence of the same variables in the main flow chart and in the subroutine.

When we have a function of more than one variable, the name of the function is accompanied by a list of variables. This list (which reduces to one entry for a function of a single numerical variable) is called the "parameter list." Parameter lists appear both in the funnel of the reference flow chart and whenever the function is referred to in the main flow chart. Although the same variable need not appear in corresponding positions of the parameter lists for a given function, these lists must nevertheless match up like the fingers of a hand and the fingers of a glove. Besides being equal in number, the variables in the lists must match as to type (numeric, alphanumeric, etc.). A failure to match as to type is like trying to put a left-hand glove on a right hand--it doesn't work.

Composition of functional references

Suppose we have two functions F1(x) and F2(x) each with its own definition in the form of a functional reference flow chart.

Now we examine the assignment box below.

$$\longrightarrow \boxed{Y \leftarrow F1(F2(T))} \longrightarrow$$

We see that in assigning a value to Y there are two intermediate steps and a final step: These are

1. Employ the functional reference flow chart for F2(x), supplying the value of T to be matched with x (i.e., dropping the value of T into the funnel). Take the value returned by F2(x), i.e., as read through the RETURN window of F2(x), and use it in the next step.
2. Employ the functional reference flow chart for F1(x), supplying the value obtained in Step 1 to be matched with x, i.e., drop the value obtained from Step 1 into the (new) funnel.
3. Take the value returned by F1(x), ie.., as read through the RETURN window of F1(x), and assign it to Y.

Take note that all along there has been the implicit assumption that the range of F2 is a subset of the domain of F1.†

With this concept of a function of a function, each function being thought of as a separate reference flow chart, we should have no trouble in interpreting the following flow chart boxes:

$$\longrightarrow \boxed{Y \leftarrow MIN(|A + B|, 5.4)} \longrightarrow$$

$$\longrightarrow \boxed{Y \leftarrow MIN(MIN(F, |T|),Q)} \longrightarrow$$

$$\longrightarrow \boxed{Y \leftarrow MIN(SQROOT(B^2 - 4AC), -B)} \longrightarrow$$

$$\longrightarrow \boxed{Y \leftarrow SQROOT(MIN(X, Y))} \longrightarrow$$

---

†You will recall that the range of a function is the set of all the numbers that the function is capable of producing as functional values.

Exercises 5-3 Set A

1. (a) Draw a flow chart for function $f(x,y)$ where $f(x,y) = \frac{(x^3 + y)^2 + 5}{|x| + 2}$.

   (b) Draw the assignment box that computes $Z = f(r,s) + 6t$ where r, s, and t have been previously assigned values.

2. Prepare a flow chart to evaluate the function

$$\text{right }(a,b,c) = \begin{cases} 1, & \text{if } a, b, c \text{ are lengths of sides for a right triangle.} \\ 0, & \text{if they are not lengths for a right triangle.} \end{cases}$$

3. (a) Prepare a reference flow chart which assigns to $\max(x,y,z)$, the largest (algebraically) of the three values of x, y and z.

   (b) Prepare a flow chart to read in values of A, B and C, calculate max(A,B,C), assign this value to LARGST, and then print LARGST.

4. Given values of x and y, the function QUAD(x,y) is to return an integer value 1, 2, 3 or 4 according to the quadrant in which the point (x,y) lies. Prepare a reference flow chart for QUAD(x,y).

5. Prepare a flow chart for function INTSCT which takes values of X1, Y1, R1, X2, Y2, R2 and returns the number of points of intersection that the circle with center (X1,Y1) of radius R1 has with the circle with center (X2,Y2) of radius R2. Allow for the possibility that R1 or R2 is accidentally given with a negative value.

6. For the n^{th} root, an iteration formula corresponding to the cube root formula given in Exercises 5-1 is

$$h \leftarrow \frac{1}{n}((n-1) \times g + \frac{y}{g^{n-1}}) .$$

   For larger values of n, the root may be approached very slowly. For this reason you should not let your computation go beyond ten iterations. Prepare the reference flow chart.

7. Jack Armstrong plans to borrow $100 and wants to compare various loan plans.

   (a) Company X will lend him the money at compound interest at 1% monthly. That is, each month 1% of what Jack owes is added to the debt he must someday repay. Draw a reference flow chart IRATE (n, R, L) which computes the amount that must be paid after n months for a loan of L dollars at R percent monthly interest.

   (b) Jack finds that Company Y will lend him money at simple interest at $1\frac{1}{8}$% per month. At simple interest Jack pays the interest monthly instead of it being added to the balance owed. Draw a flow chart to compare the amount Jack pays to Company X and Company Y after 12 months, 24 months, 36 months and finds the first month (if any) when Jack's total debt to Company X would be more then his total payment and debt to Company Y.

   (c) Jack finally wonders how long it will take each company to double its money. Find that period for both companies.

Exercises 5-3 Set B

1. Redraw the flow chart for the Euclidean Algorithm Figure 3-14 as a reference flow chart. (Is Box 2 needed in the new flow chart?)

2. Draw a reference flow chart for finding the GCF (greatest common factor) of three non-negative integers. [Note that if X = GCD(A,B), then GCF(A,B,C) = GCD(X,C). Hence use the preceding problem.]

3. (a) In Section 4-4 you constructed a flow chart for finding the number of triangles having sides whose lengths are integers not greater than 100. Now make a flow chart for solving the same problem with the added restriction that no two of the triangles shall be similar. [Hint: Use the preceding problem.]

   (b) Modify (a) so as to output instead the sum of the perimeters of the triangles described in (a).

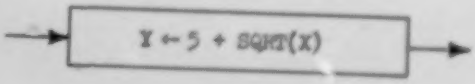
*4. Draw a flow chart to output all numbers less than 10^9 which are expressible as the sum of two cubes in two different ways together with the two decompositions into the sum of cubes. [E.g., since $12^3 + 1^3 = 1729 = 10^3 + 9^3$ the numbers 12, 1, 1729, 10, 9 would be output.] Eliminate all proportional combinations such as $24^3 + 2^3 = 18832 = 20^3 + 18^3$.

---

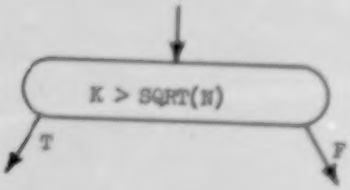
* This problem can be solved by brute force, testing every set of four cubes, requiring 10^{12} loops. You should try to find a more efficient way requiring about 10^9 loops. Finding this more efficient method will require some ingenuity.

5-4 Procedures

A functional subroutine is used when the functional value is a single numerical value. This value is assigned to the variable named in the return box of the reference flow chart. Here this value is picked up (read) by the main flow chart and employed in the appropriate computational step, such as in



or as in



As we have already seen, there are many important functions whose functional values are vectors. Such functions are called vector-valued functions. We shall need subroutines for evaluating such functions. These subroutines are called procedures. As will be seen, the operation of a procedure is different from that of a functional subroutine in several respects.

We give an example of a reference flow chart for a procedure in Figure 5-16. This flow chart is for a sorting process which is still different from those considered in Chapters 3 and 4.

We see that the procedure reference flow chart again has a funnel in which appears the name of the procedure and a parameter list. There is a return box, but with no variable in it. With the experience you have had by now it should be no trouble for you to check that this flow chart will take values assigned to $\{a_i, i = 1(1)n\}$ and sort them so as to be indexed in increasing order.

But how is the main flow chart to call for such a procedure to be executed? The situation here is unlike a functional reference where a single numerical value is to be returned and employed in a computational step. Here a vector of values is to be "returned" instead.

What is the "output" of the sort procedure? How is the main flow chart affected? These are the questions we will answer next.

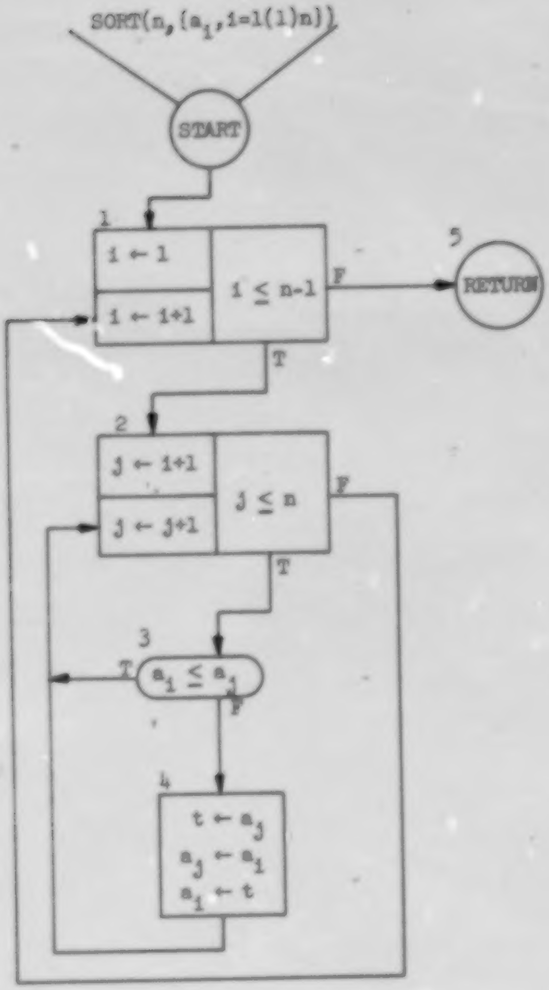
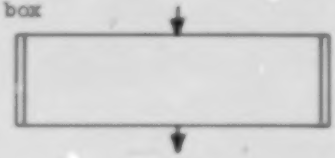

SORT(n, $(a_i, i=1(1)n)$)
START
1
$i \leftarrow 1$
$i \leftarrow i+1$
$i \leq n-1$
F
5
RETURN
T
2
$j \leftarrow i+1$
$j \leftarrow j+1$
$j \leq n$
F
T
3
T
$a_i \leq a_j$
F
4
$t \leftarrow a_j$
$a_j \leftarrow a_i$
$a_i \leftarrow t$

Figure 5-16. Reference flow chart for sort procedure

First we point out that while a call for a functional evaluation always comes during a calculation step, this is never the case with a procedure. Instead, we have a special box



containing the name of the procedure and a parameter list not usually composed of the same variables as that in the funnel but matching it item for item like fingers and gloves. And the whole works is capped by the word "Execute."

Here in Figure 5-17 we see a typical call for the sort procedure of Figure 5-16.

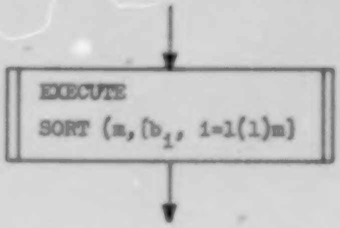


Figure 5-17. Call for sort procedure

The effect of this sort procedure on the main flow chart will be to take the values of the vector b and reshuffle them so as to be indexed in increasing order. For example, Table 5-1 shows the "after" values of the vector b.

Table 5-1

	b_1	b_2	b_3	b_4	b_5	b_7	b_8
before	7	9	2	1	6.3	-1.5	2
after	-1.5	1	2	2	6.3	7	9

This exhibits another difference between the effect of procedural- and functional- flow charts. Namely, that a procedure does change the values of the variables in the main flow chart.

But how shall we visualize the way in which the procedure is carried out, the coupling of the main flow chart and the subroutine. The description given below in terms of the old reliable window boxes is in very close analogy to what actually goes on in the computer.

Again as with the functional subroutines we have the sealed brick chamber in which the procedure is executed. But this time one thing is missing and one thing is added. There is no window for reading the "functional value" but in its place there is a long chute coming out from the side.

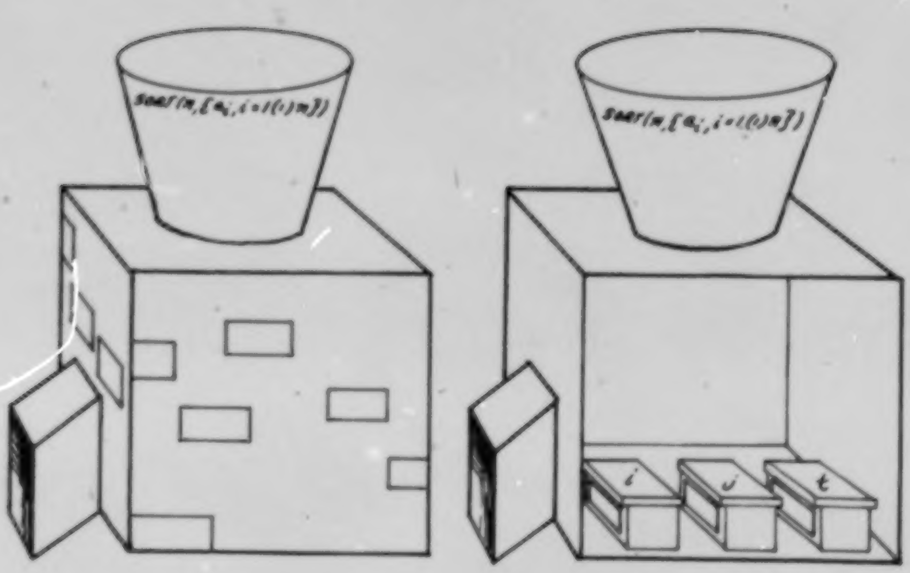
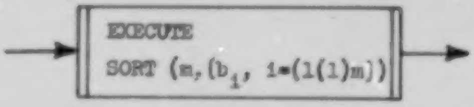


Figure 5-18. Exterior and interior views of sort procedure of Figure 5-16

In the interior of the sealed chamber we are somewhat surprised to see window boxes only for the local variables, none for the non-local variables. That is, there are no window boxes for the variables which appear in the parameter list in the funnel. [Note that in the illustrated case, the variable i is not considered to be in the parameter list. This list is considered to consist of n, a_1, a_2, a_3, a_4, etc., with the last subscript being the value of n.]

In the example under discussion, only boxes for i, j and t will be found inside the brick chamber. And now when the procedure is called for by this box in the main flow chart



a surprising thing happens. Instead of the values of the variables m, {b_i, i = 1(1)m} being put in the funnel, the window boxes belonging to these variables are brought to the funnel and dropped in, in the order in which they appear in the parameter list. (Figure 5-19).

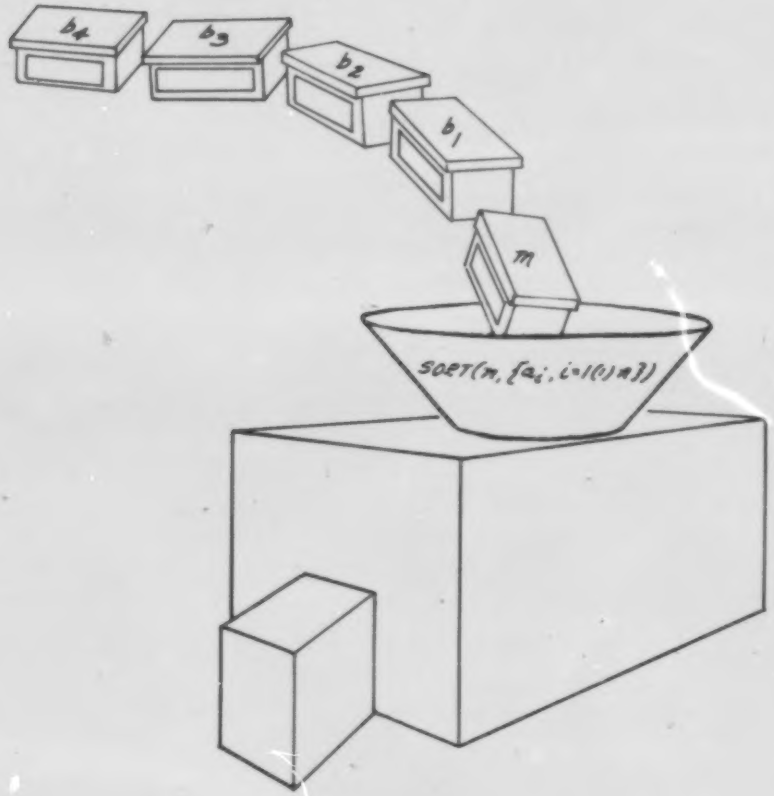


Figure 5-19. Main flow chart window boxes coming to procedures

As these boxes pass through the funnel they are received by a specialist of the procedure staff, the labeler. The labeler has a stack of peelable (removable) gummed stickers. These stickers are imprinted in order with the variables of the parameter list appearing in the funnel of the procedure. As the boxes come to the labeler he slaps a sticker on each, covering up the original inscription on the box. (Figure 5-20).

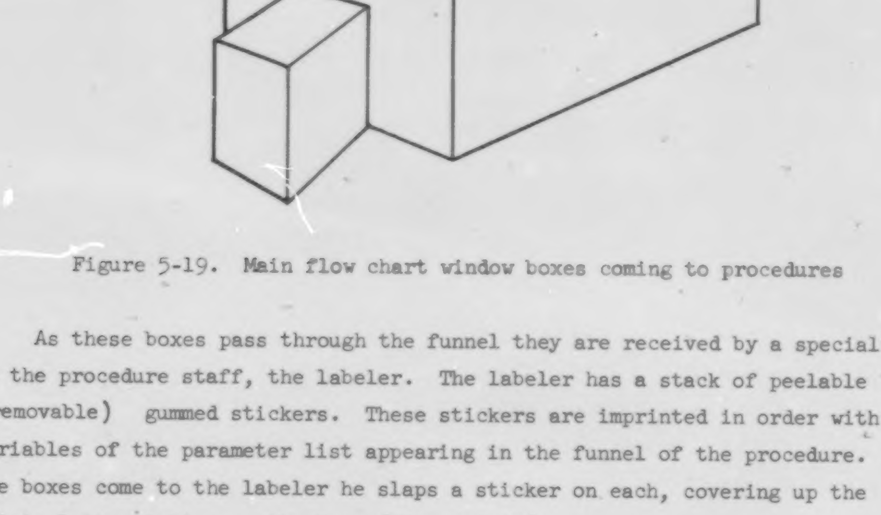


Figure 5-20. The labeler at work

When the last window box has been labeled the chamber is sealed so that there is no further contact with the main flow chart. The interior of the procedure of this stage is as in Figure 5-21. For this figure we are assuming the value of m equals 11.)

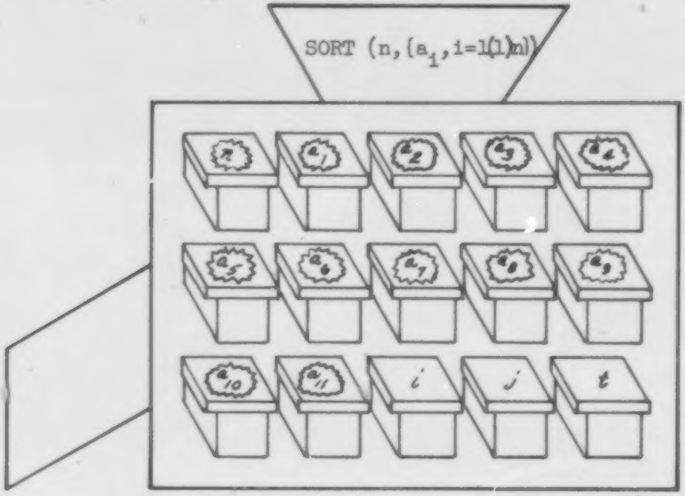


Figure 5-21. Interior of procedure after execution is called for

Now the procedure is executed exactly as it appears in the procedure flow chart. The assigner and the reader operate using the variables on the stickers without even knowing what the original variables on the boxes were. The initial values in these boxes are of course those they brought with them from the main program.

When the execution of the procedure has been completed and we come to the return box, what then? Well, at this point the door to the chute opens and all the boxes with the gummed stickers are dumped out. Here another specialist, the unlabeler, peels off the gummed stickers and the boxes return to the main program bearing their original inscriptions. (Fig. 5-22).

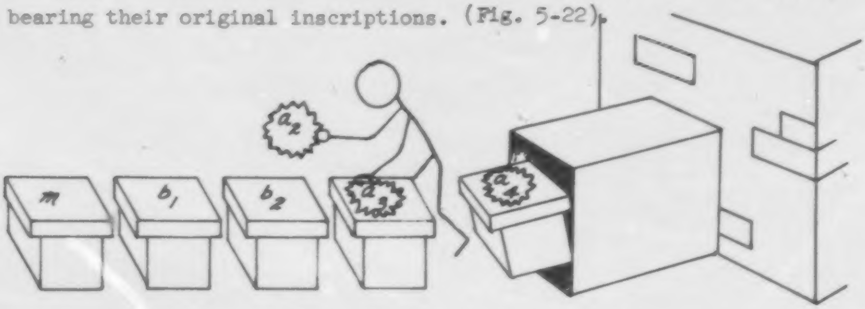


Figure 5-22. The unlabeler at work

Now we can summarize the net effect of the procedure. A certain well-defined list of main flow chart variables has "fallen under the sway" of the reference flow chart and some, or all, of these variables may have had their values changed in the process.

An example of a main flow chart which uses the "sort" procedure is given in Figure 5-23. The flow chart there will cause a number to be input and assigned to k. Then $2k$ numbers will be input and assigned to $b_1, b_2, \ldots, b_k$, $c_1, c_2, \ldots, c_k$. For the first use of the sort procedure, the variables k, $b_1, b_2, \ldots, b_k$ are substituted for n, $a_1, a_2, \ldots, a_n$ in the definition of the procedure. Then the steps in the procedure itself are carried out. After returning to the main flow chart the value of k will be unchanged, but the values of $b_1, \ldots, b_k$ will have been rearranged into non-decreasing order.

The second use of "sort" will cause the c's to be rearranged.

Finally, the b's and c's are output, the smallest b followed by the smallest c, then the next smallest b, followed by the next smallest c, and so on.

It should be recalled that the local variables in Figure 5-16 are i, j and t.

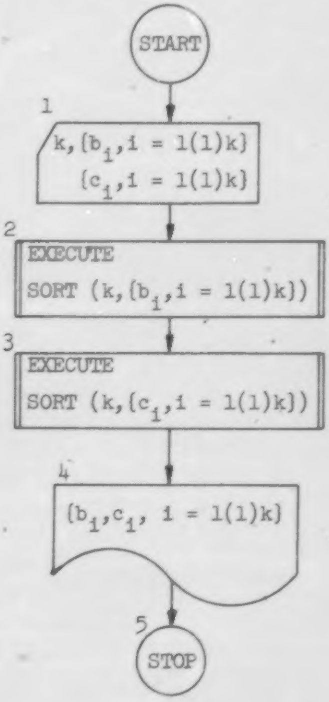


Figure 5-23. An example of execute boxes

Exercises 5-4 Set A

1. Rewrite the flow chart of function ABSOL(X) from Exercises 5-1 Problem No. 4, to make ABSOL(X) into a procedure.

2. (a) For complex numbers of the form a + bi, prepare a reference flow chart for procedure cxadd(a1,b1,a2,b2,a,b) which accepts the real parts a1 and a2, and the imaginary parts b1 and b2 of two complex numbers and returns their sum (a,b).

   (b) Similarly, prepare a reference flow chart for procedure cxsubt which computes the difference of two complex numbers: (a1 + b1i)-(a2 + b2i).

   (c) Prepare a procedure flow chart for cxmult which yields the product of two complex numbers.

   (d) The quotient of two complex numbers is to be given by procedure cxdiv. Prepare the reference flow chart. Take (a1,b1) as the dividend.

   (e) Prepare a flow chart which inputs two complex numbers and an index oper which indicates the operation (1 for add, 2 for subtract, 3 for multiplication, 4 for division) to be performed on them. The output will be an echo check of input followed by the result of the computation printed in the form a "+" b "i". Then read a new set of data.

3. Prepare the reference flow chart for a procedure called SORT2. The parameter list is to be $(K, \{A_i, i = 1(1)K\}, \{B_i, i = 1(1)K\})$. $\{A_i\}$ is a vector whose values are to be arranged in non-decreasing order and $\{B_i\}$ is a vector to be rearranged in the same way as $\{A_i\}$, so that B_J remains associated with A_J. Assume $K < 10^3$.

4. (a) Prepare a reference flow chart for a procedure that receives a given integer N and returns COUNTFAC, the count of integer factors which N has. (Hint: You may wish to adapt Figure 4-12.) Could this procedure have been written as a function?

   (b) Prepare a flow chart which uses COUNTFAC to output a list of all prime numbers up through 1000. Of course, a prime number is an integer having only two factors, 1 and the number itself. Explain why this is a very inefficient method for making a list of primes.

5. (a) Prepare a flow chart for a procedure aliquot(number,PARTS,n) which takes a given integer number, less than 10^3 and returns its aliquot parts in the first n elements of vector PARTS. The aliquot parts of a number may be thought of as being all its factors except the number itself; e.g., the aliquot parts of 12 are 1, 2, 3, 4, 6.

   (b) A "perfect" number was considered by the Greeks to be a number having a value equal to the sum of its aliquot parts; e.g., for 6, $1 + 2 + 3 = 6$. Prepare a flow chart for finding all perfect numbers up through 500.

   (c) 220 and 284 are referred to as "friendly" numbers since the aliquot parts of 220 total 284, and the aliquot parts of 284 total 220. Prepare a flow chart for finding all friendly numbers up through 500.

Exercises 5-4 Set B

1. Draw a functional reference flow chart (LEAST) to find the algebraically smallest element of the vector A with components $A_1, A_2, \ldots, A_N$.
2. Draw a functional reference flow chart (SUBLEAST) to find the subscript of the smallest element of A.
3. Draw a reference flow chart for a procedure called MARKS to find both the smallest element of A and its subscript. Can a single functional reference flow chart be used instead?

Exercises 5-4 Set C

1. Recall Problem 8, of Section 3-5, Set A, to determine the actual degree of a polynomial given n and the set of coefficients $\{a_i, i = 0(1)n\}$. For this problem, assume the coefficients are integers and draw the flow chart for a procedure called DEGREE which accomplishes the same thing as Problem 8. That is, another program can call on DEGREE by supplying it with the polynomial data n, $\{a_i, i = 0(1)n\}$ as arguments. DEGREE returns control to the calling program when it has revised (or "put it's stamp of approval on") the value of n. Assume the apparent degree is < 100.

2. Draw the flow chart for a procedure SIMPLIFY, for taking a polynomial with integer coefficients, represented by $n, \{a_i, i = 0(1)n\}$, and replacing it by the polynomial obtained when one divides each coefficient by the greatest common factor of the coefficients. (Hint: Use the GCD function reference flow chart.)

*3. If $a(x)$ and $b(x) \neq 0$ are polynomials with integer coefficients, then there exist a non-zero constant c, and polynomials $q(x)$ and $r(x)$ with integer coefficients and with degree $r(x) <$ degree $b(x)$ so that

$$c \cdot a(x) = q(x) \cdot b(x) + r(x) .$$

This is called the remainder theorem for polynomials. The process of finding $r(x)$ is referred to as reducing $a(x)$ **mod** $b(x)$.

(i) Draw the flow chart for a procedure called REDUCEMOD which takes two integer polynomials $a(x)$ and $b(x)$ represented by $n, \{a_i, i = 0(1)n\}$ and $m, \{b_i, i = 0(1)m\}$ where $b(x)$ is known to have actual degree m and to be already simplified as in Problem 1, and

(ii) computes $r(x)$ as in the above formula, simplifies $r(x)$ as in Problem 1, and replaces $a(x)$ by $r(x)$. Be sure that no fractions occur in the computation so that there will be no round-off error.

(Hint: Use the procedures DEGREE, SIMPLIFY and GCD.)

*4. Draw a flow chart for finding the greatest common divisor in simplest form of the two polynomials with integer coefficients. By a greatest common divisor of two polynomials we mean a polynomial of highest possible degree which is a divisor of both the given polynomials, the quotients being permitted to have fractional coefficients. Parrotting the derivation of the Euclidean algorithm in Section 3-2 we find that this greatest common divisor is unique except for multiplication by any non-zero rational number. By "simplest form" we mean that the coefficients must be integers having no common factor. It is quite evident that multiplying the given polynomials by non-zero integers will not alter this simplest greatest common divisor. (Hint. Use the procedures DEGREE, SIMPLIFY and REDUCEMOD.)

---

*These problems are quite difficult. The student will be able to solve them only with considerable time and effort.

5-5 Extensions to Reference Flow Charts and their Models

Recall that in the window box model, we always drop numerals written on slips of paper into the funnel for functionals and we drop the window boxes themselves into the funnel for procedures. What would happen if we were to drop slips of paper into the procedure funnel? Would there be any advantage in being able to do this?

In the first case (slips of paper) we do not want what goes on inside the sealed chamber to directly affect what is in any window box on the outside. In the second case (boxes) the action within the chamber can affect the contents of window boxes which normally "reside" outside the procedure (chamber).

Using slips of paper or using window boxes actually corresponds to giving a reference flow chart the data itself or to giving the address in memory where the data can be found. If the reference flow chart knows the address of a variable A, it can change the value of that variable. If it only knows the value of A but doesn't know where it came from, it can use that value but cannot change it in memory.

If we want to be sure that a variable in the parameter list of a procedure remains unchanged no matter what happens inside that sealed chamber, we should drop in the slip of paper and not the window box containing that slip of paper. As suggested in Figure 5-24, when a slip of paper flutters in instead of a window box, the assistant puts it in the window box corresponding to the appropriate (non-local) variable of the procedure. Remember that such a variable is called non-local because it can have values assigned to it from the outside. It's window box always remains inside the brick chamber.

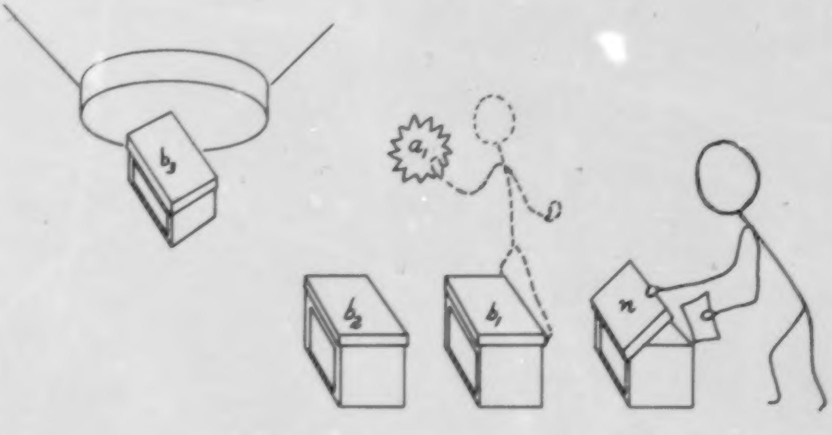


Figure 5-24. Assistant assigning to a variable

Now let us ask the converse question. Would there be any advantage in being able to drop window boxes down the funnel of a functional subroutine? According to our model there would be no way of getting the boxes back out. The situation is summarized in Figure 5-25. This could be fixed by attaching a chute leading from the chamber but then the real distinction between a functional subroutine and a procedure is lost.[†]

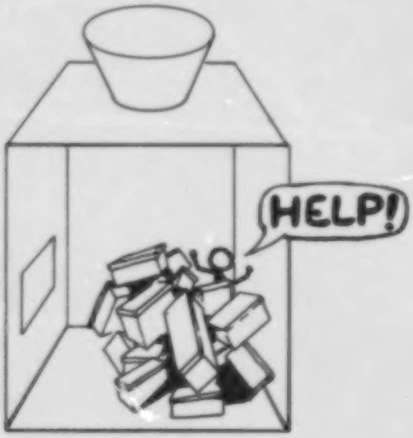


Figure 5-25. Life imprisonment in a functional subroutine

Alternate exits

There are frequently situations in which we could use a procedure to indicate (perhaps in addition to a calculated result) which of two or more paths the main flow chart should follow. In some reference flow charts, there are alternate paths to be pursued because unusual (or error) situations may arise. For other reference flow charts the selection of alternate paths may be the principal purpose of the procedure.

One technique for choosing alternate paths is to include an output variable whose value will tell the main flow chart which path to take. This technique is illustrated in Figure 5-26 for a procedure with the purpose of determining whether two complex numbers $a + ib$ and $c + id$ are equal.

[†] Some programming systems do, in fact, provide addresses (i.e., the boxes) to functional subroutines. If the system you are using does do this you should know about it since assigning values to such variables in a functional subroutine will yield different results depending on the programming system.

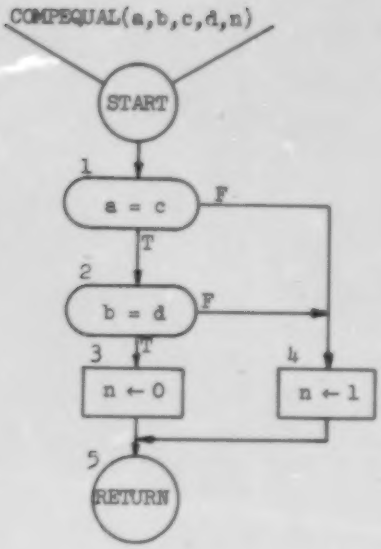

COMPEQUAL(a,b,c,d,n)
START
1
a = c
F
T
2
b = d
F
3
T
4
n ← 0
n ← 1
5
RETURN

Figure 5-26. Test for equality of complex numbers

From this figure we see that the only effect of the procedure is to assign values of zero or one to the variable n depending on whether the complex numbers are or are not equal. Since n is the output variable, it appears in the parameter list. (Could this flow chart have been prepared as a functional reference instead of a procedure?)

Figure 5-27 shows how this "compequal" procedure could be used in a main flow chart to decide whether or not the complex numbers x + iy and u + iv are equal.

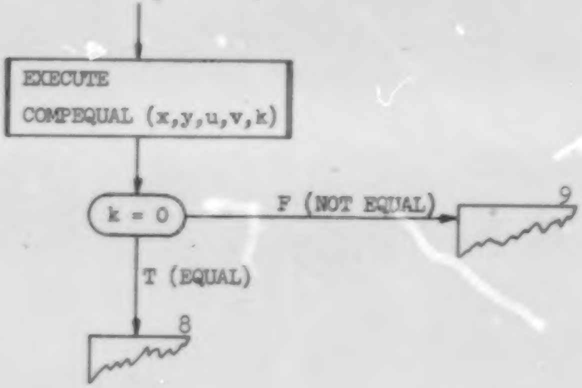

EXECUTE
COMPEQUAL (x,y,u,v,k)
k = 0
F (NOT EQUAL)
9
T (EQUAL)
8

Figure 5-27. Are x + iy and u + iv equal?

A second technique for using a procedure to choose alternate paths is that of actually combining the EXECUTE box and the decision box. In the flow chart language this can be pictured as in Figure 5-28.

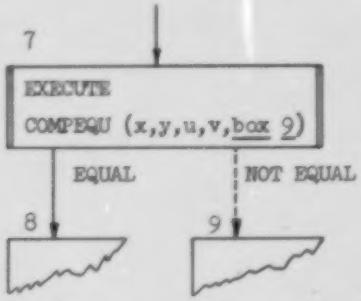

7
EXECUTE
COMPEQU (x,y,u,v,box 9)
EQUAL
NOT EQUAL
8
9

Figure 5-28. Choosing alternate paths

The parameter list of COMPEQU contains, in addition to variables for the real and imaginary parts of the numbers to be compared, the flow chart box number identifying the start of an alternate path in the flow chart. We deliberately underscore the flow chart box number (procedural languages often called it the statement label) to distinguish it from a variable. In Figure 5-28 the normal return leads to box number 8 while an alternate return leads to box number 9. A reference flow chart for this procedure is shown in Figure 5-29.

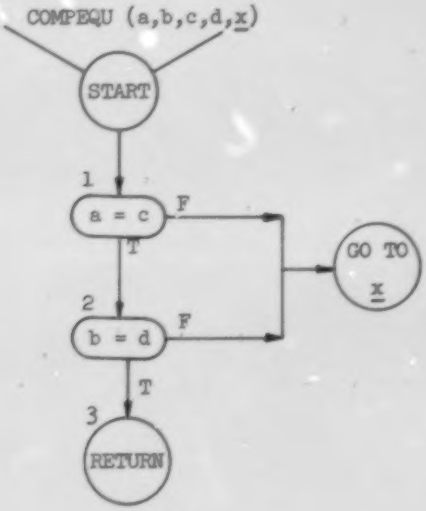
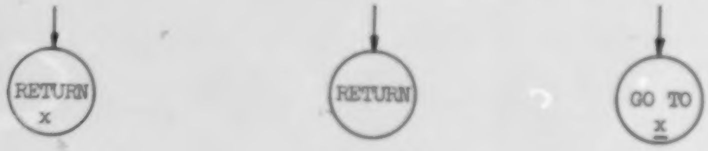

COMPEQU (a,b,c,d,x)
START
1
a = c
F
T
GO TO
x
2
b = d
F
T
3
RETURN

Figure 5-29. A procedure to make a choice

Compared with Figure 5-26 we see that boxes 3 and 4 of that figure have been eliminated. A new terminal (we always use circles for the terminals of our flow charts) appears with the direction "GO TO $\underline{x}$". This is an alternate exit from the procedure. We now have three distinctly different ways of ending a reference flow chart, each represented by a distinct box as shown in Figure 5-30.



(a) Return a value (b) Normal exit (c) Alternate exit

Figure 5-30. Comparison of reference flow chart endings

Form (a) is used only in functional reference flow charts to indicate that the value of a local variable x is to be returned to the calling program. Form (b) is used in reference flow charts for procedures to indicate that the procedure has been completed and the next flow chart box in normal sequence is to be used next. Form (c) is used in reference flow charts for procedures to indicate that the procedure has been completed and the next flow chart box to be used is $\underline{x}$.

The window box model of our procedure needs a new feature to reflect the GO TO terminal box. It is pictured in Figure 5-31.

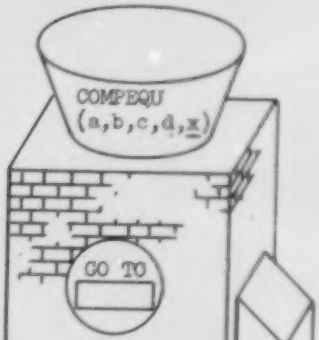
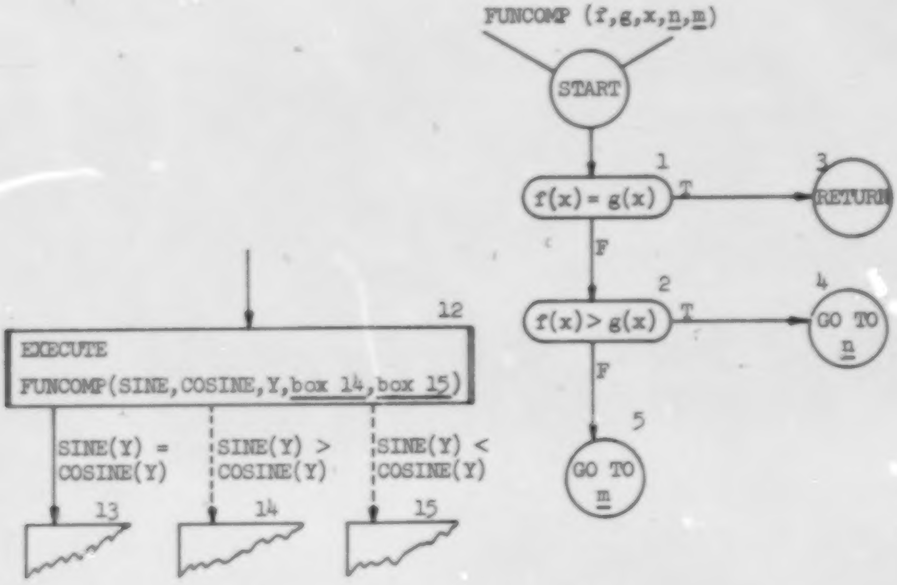


Figure 5-31. A choice procedure

When the master computer wants to use COMPEQU he sends in the window boxes inscribed with x, y, u, and v and in addition a slip of paper on which is written box 9 (the label of the alternate next flow chart box). Four boxes and the slip of paper are dropped in the funnel in the proper order; gummed labels (a, b, c, d) are stuck on the four boxes in order and the slip of paper is assigned to the local window box x. Now the busy crew inside the sealed chamber gets to work until finally the four boxes slide out of the chute. If the crew determines that the alternate exit is to be used, they shove window box x up against the GO TO window. In collecting these boxes, the reader and his crew now check the "GO TO" window to see whether it is empty or contains the label of a flow chart box. The reader makes a note of whatever is in the window and returns it to the master computer who now knows which flow chart box is next.

Function Name Arguments

It will be valuable to be able to use the name of a reference flow chart (either a functional reference or a procedure) as an argument of a procedure. Suppose, for example, that we want a procedure to compare the values of functions f(x) and g(x) for a given value of x. Figure 5-32 shows how such a procedure can be flow charted.


FUNCOMP (f,g,x,n,m)
START
1
f(x) = g(x)
T
3
RETURN
F
2
f(x) > g(x)
T
4
GO TO n
F
5
GO TO m
12
EXECUTE
FUNCOMP(SINE,COSINE,Y,box 14,box 15)
SINE(Y) = COSINE(Y)
SINE(Y) > COSINE(Y)
SINE(Y) < COSINE(Y)
13
14
15

(a) EXECUTE box

(b) Reference flow chart

Figure 5-32. Procedure to compare functional values

In Figure 5-32(b), dummy names for two functions, f and g, appear as "formal parameters" in the parameter list of FUNCOMP. The corresponding "actual parameters" (in this example, the function names SINE and COSINE) are in the parameter list of FUNCOMP in Figure 5-32(a). In addition, FUNCOMP has two alternate exits (the formal parameters n and m corresponding to the actual parameters box 14 and box 15).

When FUNCOMP is called upon for execution, the actual parameters are associated with the formal parameters of the reference flow chart. That is, FUNCOMP is instructed that, for this execution, f is to be considered the sine function and g is to be considered the cosine function.

We are sure that the student will think of ways to extend the window box model so that function names can be used inside a sealed chamber for a procedure. We will leave this final extension of the model up to your imaginations.

Exercises 5-5

1. Draw a flow chart of a procedure for solving two equations in two unknowns. Prepare an alternate exit for the special case when the two equations correspond to parallel or concurrent lines.

2. (a) Draw a reference flow chart for a procedure to find the real roots of any quadratic equation (see diagram in Chapter 1). Your flow chart is to use alternate exits to distinguish finding of no roots, one root, two real roots or imaginary roots.

   (b) Draw a flow chart which will use the reference flow chart of part (a) and will print the roots found with an appropriate message.

   (c) Draw a flow chart to do the same job as the flow chart of part (a) but without using alternate exits. You must provide a variable which will carry back to the main flow chart the formation as to which of the four possible cases has occurred.

   (d) Draw a flow chart which will use the reference flow chart of part (c) and print out the roots found with messages as in 2(b).

3. Refer to Problem 1, Section 5-3, Set A. Suppose the function f were redefined as

$$f(x,y) = \frac{(x^3 + y)^2 + 5}{x - 2} .$$

   This function is not defined at $x = 2$.

   Draw two procedures for computing values of f, one using an alternate exit, the other using a variable to carry the information about the exceptional case back to the main flow chart. In each case show a flow chart fragment which calls for f and computes

$$Z = f(r,s) + 6 \times W .$$

4. It is desired to sum up the values of a function at a number of equally spaced points in an interval starting with the left endpoint. Draw a flow chart for a procedure which will carry this out for any function, any interval and any spacing (or increment). Provide an error exit for the case that the increment is negative or that the number given as the right endpoint of the interval is to the left of the left endpoint.

5-6 Character strings

Problems in an area which is often called "symbol manipulation" will be considered in Chapter 8. Special procedures are needed to handle these problems. In this section we will develop some of these special procedures.

We begin by assuming that we have a string of characters, which we will denote by $s_1, s_2, \ldots, s_n$. These characters could be numbers, but in the area of immediate interest we will think of them as being mainly alphabetic characters. We will include punctuation marks, spaces, and digits, as well as all the letters of the alphabet.

A string might therefore represent a sentence. For example, the string might be: The quick brown fox jumped over the lazy dog. In this case s_1 is T, s_2 is h, s_3 is e, s_4 is a space, s_5 is q, and so on.

A string could also represent the digits and decimal point of a number such as 3.14159. In this case s_1 is 3, s_2 is the decimal point, s_3 is 1, and so on.

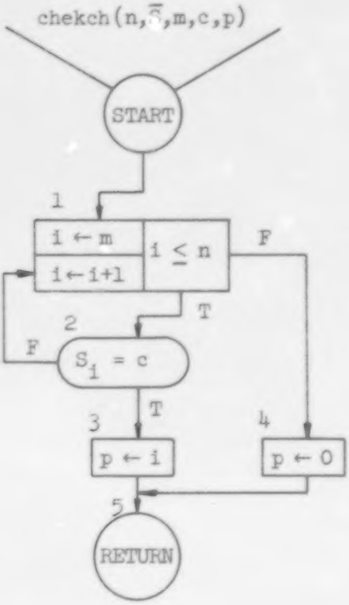
A string could also represent a mathematical expression such as $r + s(t + u(v + w))$. Here s_1 would be r, s_2 the plus sign, etc.

To begin with we will need notation for convenient reference to strings, elements of strings and substrings. Subscripted lower case letters will be used to identify individual elements of strings as in the last few paragraphs. A string (or a substring) can be referred to by the familiar looking notation, $\{s_i, i = 1(1)n\}$ which is read as "the set of elements s_i, with i varying from 1 to n in steps of 1. Such notation is particularly convenient for reference to substrings. For example, $\{s_i, i = 3(1)8\}$ refers to the third through eighth (inclusive) elements of a string. On the other hand, such explicit notation becomes cumbersome when complete strings are necessary. If the length of a string is n, $\{s_i, i = 1(1)n\}$ refers to the complete string but we will also use a shorthand and call this string $\overline{S}$ (a capital letter with a bar (or string) over it).

You are undoubtedly wondering what the distinction is between a string and a vector. Although we usually think of the components of a vector as having numeric values and the elements of a string as being characters, this is not the basic difference. The real difference is that operations on vectors do not normally change the number of their components; operations on strings frequently change the number of their elements.

In problems concerned with such strings there are certain basic procedures one would like to have available. One of the simplest of these is a procedure for searching a string for a particular character. For example, we might wish to find the first space in a sentence, or the decimal point in a number, or the first right parenthesis in a mathematical expression.

A procedure for carrying out this task is shown in Figure 5-33. The name of the procedure is chekch (for "check for character"). As input variables it has the length of the string, n, and the name of the string, $\overline{S}$. Another input variable is denoted by m; the search for the required character is supposed to begin at s_m. This feature makes the procedure much more useful than if the search always began with s_1. Then the final input variable has as its value the particular character of interest, and it is denoted by c. The only output variable is p, which stands for the subscript at which the character represented by c is first found. The search for this character is begun with s_m, then if s_m is not equal to c, s_{m+1} is tried, and so on. The first appearance of the character value of c is then s_p. The understanding will be that p is set equal to zero if c is not equal to any of



n = length of string

$\overline{S}$ = string

m = subscript of first element to be examined

c = character variable

p = subscript at which character is found

Figure 5-33. Searching for a character

the substring $s_m, s_{m+1}, \ldots, s_n$.

We can expand the idea of the chekch procedure if, instead of searching a string for a single character, we consider the problem of searching a string for a specified substring.

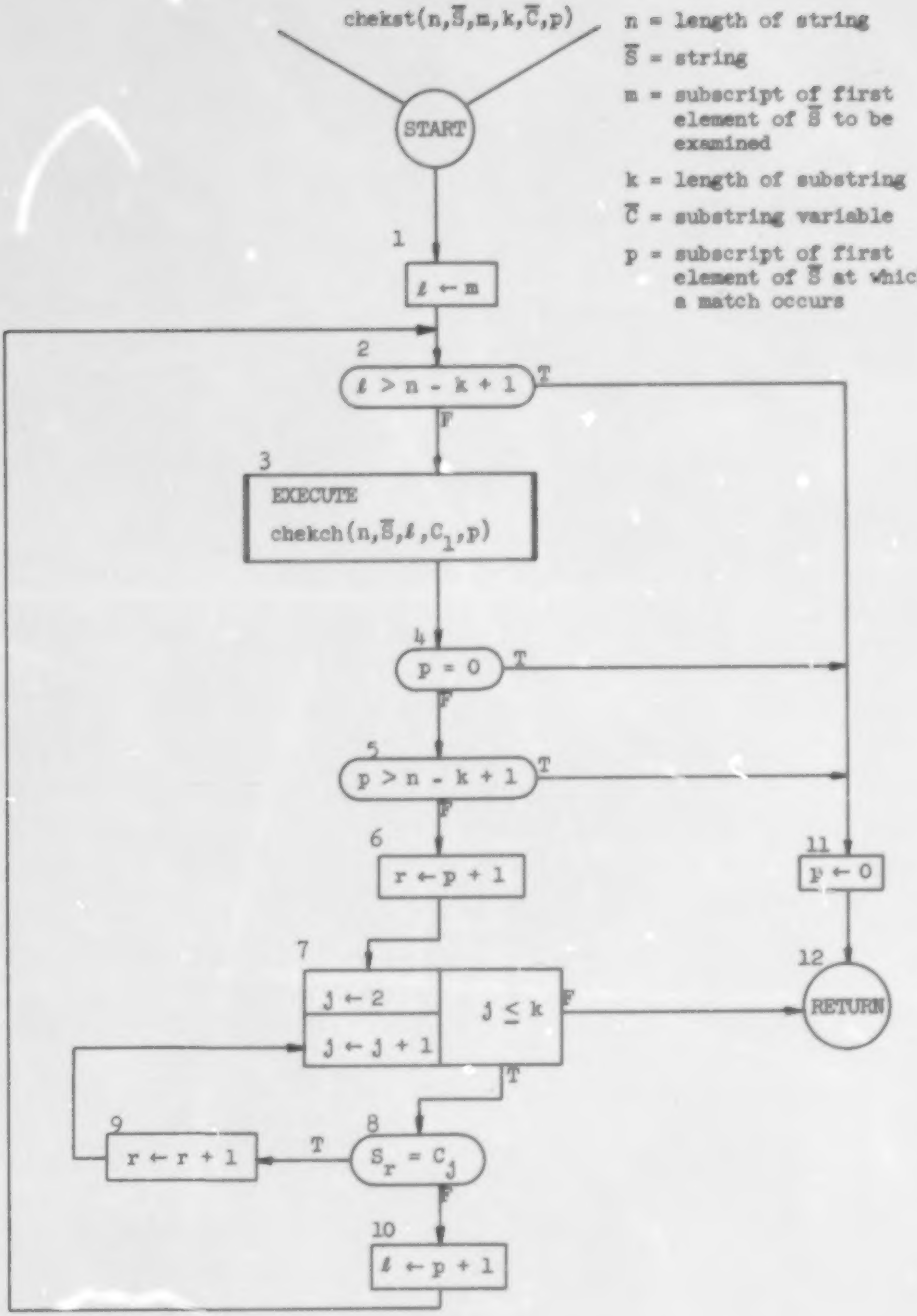
Suppose our given string is a sentence, such as: The quick brown fox jumped over the lazy dog. Then our new procedure could be used to find the number of occurrences of the substring which consists of a particular word, such as: the.

The procedure is given in Figure 5-34. The name of the procedure is chekst. The variables are the same as in the preceding section except that the single variable c is now replaced by the length k of the specified substring, followed by the name of the substring itself, $\overline{C}$. If p is not set equal to zero, then $s_p, s_{p+1}, \ldots, s_{p+k-1}$ is the first occurrence of the substring in which $p \geq m$.

On entry to the procedure the variable ℓ is given the value of the starting point m. Of course, there is no point in looking for the first appearance of c_1, beginning with s_ℓ, if this starting point is too far along the string, i.e., if $\ell > n-k+1$. If $\ell \leq n-k+1$ the procedure chekch is used to find the first occurrence of c_1, beginning the search at s_ℓ.

Then, if there is no occurrence at all, p is set equal to zero, and there is nothing to do but return. If there is an occurrence, but the value of p at which it occurs is too large, then p must be set equal to zero before returning.

If there still has been no return from the procedure, the next $k - 1$ characters $s_{p+1}, s_{p+2}, \ldots, s_{p+k-1}$ must be compared with $c_2, c_3, \ldots, c_k$. If there is agreement, the procedure returns, the value of p being correct as it stands. But if any pair do not agree, the entire process must be started over again, but this time with the new starting point at s_{p+1}. Accordingly, ℓ is assigned the value $p+1$, and the process is repeated, except, of course, for the initial assignment to ℓ.


chekst(n,$\overline{S}$,m,k,$\overline{C}$,p)
START
1
$\ell \leftarrow m$
2
$\ell > n - k + 1$
T
F
3
EXECUTE
chekch(n,$\overline{S}$,ℓ,C_1,p)
4
p = 0
T
F
5
p > n - k + 1
T
F
6
$r \leftarrow p + 1$
11
$p \leftarrow 0$
7
$j \leftarrow 2$
$j \leftarrow j + 1$
$j \leq k$
F
12
RETURN
T
8
$S_r = C_j$
T
9
$r \leftarrow r + 1$
F
10
$\ell \leftarrow p + 1$

n = length of string

$\overline{S}$ = string

m = subscript of first element of $\overline{S}$ to be examined

k = length of substring

$\overline{C}$ = substring variable

p = subscript of first element of $\overline{S}$ at which a match occurs

Figure 5-34. Searching for a string

Figure 5-34 is a good example of one procedure being used by another as discussed in the preceding section. In many cases, up to now, we have drawn flow charts which were thought of as "main flow charts". Now, we can consider such flow charts as being procedures themselves. In this way we see how more and more complicated building blocks can be constructed.

Exercises 5-6

1. Construct a procedure to be called contch for counting the total number of occurrences of a specified character in a given string. The parameter list should be $(n, \overline{S}, c, count)$.

2. In an arithmetic expression, parentheses must "match up." That is, every left parenthesis must have a corresponding right parenthesis. Suppose that during the scan of a string representing an arithmetic expression, a counter is kept which starts at zero, adds one for each left parenthesis encountered and subtracts one for each right parenthesis encountered. Then if the expression is correctly written (with respect to parentheses), the counter must never have a negative value and must have a zero value at the end of the scan. Prepare a procedure flow chart (called parenchek) to determine if an arithmetic expression is properly parenthesized.

3. Construct a procedure called contst for finding the total number of occurrences of a specified substring in a given string. The parameter list should be $(n, \overline{S}, k, \overline{C}, count)$.

4. A teacher has the daily grades for his class for an entire term in a string $\overline{A}$. The grades are grouped by student; i.e., $a_1, \ldots, a_i$ are for the first student, $a_{i+1}, \ldots, a_j$ are for the second student, etc.
    (a) Flow chart a function called aver which will average elements m through n of string $\overline{A}$.

    (b) Write a program using aver which inputs a string $\overline{A}$, inputs the first and last subscripts, m and n, of the grades for a single student, outputs these subscripts together with the student's average, and then returns for data on the next student.

Chapter 6

APPROXIMATIONS

6-1 Introduction

In the preceding chapters of this book you have learned how to construct algorithms for the solution of a variety of problems. One of the important advantages of constructing an algorithm for the solution of a class of problems is the ability to subsequently delegate the execution of the many instructions contained in the algorithm to someone else. In particular, this someone else may be a digital computer. A computer is an obedient and docile servant of mankind, but like most servants it has certain peculiarities and idiosyncrasies which can be exasperating and sometimes lead to difficulties. In this chapter we shall explore some of the problems which arise when algorithms are executed on digital computers. The particular problems which we shall discuss arise out of the way in which computers handle numbers.

You have been accustomed to using numbers for at least as long as you have been in school and are therefore quite familiar with them. Familiarity often breeds contempt, and you may therefore not appreciate the elegance and sophistication with which you treat numbers.

In your earlier study of mathematics you have learned about the system of real numbers. From the mathematical point of view, this society of real numbers is extremely democratic in the sense that any real number is just as good as any other. However, when you have to write numerals representing these numbers, certain differences appear. The real numbers which you encountered first were whole numbers which had rather simple names such as 1, 347, 5763897. You also learned about fractions like $\frac{1}{2}$, $\frac{17}{32}$, etc. Then there were decimal fractions such as those occurring in 3.1416, 0.9823, 6.17. Finally, you learned that there are real numbers which cannot be expressed in any of the previous forms. The most popular example of this kind of number is π. At this point things get tricky. We can approximate the number π by means of ordinary fractions such as $\frac{22}{7}$, or by decimal fractions such as 3.14, or 3.1416, or 3.141592, and many others. You also learned about numbers like the square root of two, cube root of three, fourth root of twenty-six, etc., which could be approximated but not expressed exactly in any of the previous three forms.

The use of digital computers imposes restrictions on the freedom of expression of numbers. At this point review carefully Section 1-5. In particular, numbers may be expressed in two forms. The first of these is the integer form. The second is the so-called floating-point form, which consists of a series of digits with a decimal point. Numbers which are not in one of these forms have to be converted to one of these forms, or if this is not possible, suitable approximations must be found.

Even with these two forms there are limitations. A natural property of integers is that their size may be as large as you please. In a computer, however, there is always a limitation on size as the machine cannot handle arbitrarily large numbers. The maximum size of the integer which may be expressed in a computer depends on the particular machine used, and is a function of how the machine was designed. Most computers can handle integers whose expression requires up to 10 decimal digits. If larger integers are necessary in the solution of certain problems, then special steps have to be taken to accomplish the task. In the case of floating-point expressions, there are also limitations which depend on the machine used. In most cases, machines will naturally handle numerals requiring up to eight decimal digits and a decimal point. This is usually sufficient to handle the majority of problems, but again special procedures may be developed if greater accuracy is required. You may think that such a degree of accuracy is sufficient, since you probably have not had any occasion to do problems requiring greater accuracy. This is probably due to the fact that manual procedures are extremely messy with large numerals. This is one reason why computers are so useful. In the remainder of this chapter we will show you examples of the need for this kind of accuracy as well as examples in which greater accuracy is required.

Exercises 6-1

1. For each of the numerals listed below, tell whether the number can be expressed exactly in digital form (base 10) or not. If the answer is yes, express the number digitally.

(a) $\sqrt{2}$ (f) $\sqrt[3]{27}$

(b) 34.2 (g) 250,827.36

(c) 3426. (h) 0

(d) $\frac{1}{9}$ (i) $\frac{22}{7}$

(e) $\frac{5}{64}$ (j) π

6-2 Chopping and Rounding to n Digits

In Section 2-5 you studied the general problem of roundoff. In this chapter we shall use two of the types you studied. You may wish to review these briefly.

By chopping a number x to n digits we mean taking the digital representation of x, locating the first nonzero digit, and replacing by zero all digits n or more places to the right of that digit.

Examples:

x	n	Result of chopping x to n digits
3.54276	4	3.542
.0079629	3	.00796
5742	3	5740

By rounding a number x to n digits we mean the process of locating the nth digit following the first nonzero digit, increasing the absolute value of the number by 5 (for base 10) in that digit position, and then chopping to n digits.

Examples:

x	n	Result of rounding x to n digits
3.54276	4	3.543
.0079629	3	.00796
5742	3	5740
5745	3	5750

Exercises 6-2

In Exercises 1 through 4 chop the given number to 3 digits.

1. 49746 2. .007235 3. 42.37 4. 7777

5-8. Round to three digits the number given in Exercises 1 through 4, respectively.

6-3 Three Digit Arithmetic

By using a computer to execute the algorithms which you have constructed earlier in this course, you have probably learned the hard way that a computer operates with integers and floating-point numbers. Here we are primarily concerned with floating point arithmetic. Computers generally execute your programs by using a fixed word length for floating point numbers. Eight decimal digits constitute a popular word length, but some machines use shorter or longer words. We intend to show you how computers can occasionally accumulate sizable errors. If possible, algorithms should always be constructed so as to minimize errors. Since computers often carry out millions of consecutive arithmetic operations, the errors generated can become quite large. Do not be frightened. We purposely are about to show you fairly simple, and yet horrible examples illustrating what can happen. Things do not often get this bad, but you ought to be on guard against them if you are going to use computers.

To make the arithmetic relatively easy to follow, we shall work with a floating-point word length of 3 or 4 digits in our examples. The results will then be somewhat analogous to what would happen if you used 8-digit data in an 8-digit word length computer. So, if your computer does 8-digit arithmetic and you have, say, 3-digit data, then in some algorithms you may have a built-in cushion guarding against accumulation of error. But in other problems, the results are independent of word size in both computer and data, as you will see.

How does a computer do arithmetic with floating-point numbers? Since the word length is fixed, say 3 digits, each number, and each intermediate result must have 3 digits. For the correct interpretation of this last statement we must not regard a single multiplication operation as having any intermediate steps. For example, in multiplying 92.7 by .876 our three digit chop computer would first find all six digits of the product and then chop off to three digits. To illustrate:

RIGHT

$$\begin{array}{r} 92.7 \\ \underline{.876} \\ 5562 \\ 6489 \\ \underline{7416} \\ 81.2052 \end{array}$$

WRONG

$$\begin{array}{r} 92.7 \\ \underline{.876} \\ 5562 \\ 6489 \\ \underline{7416} \\ 81.0 \end{array}$$

Thus in our computer arithmetic

$$92.7 \times .876 = 81.2 .$$

In the problem

$$92.7 \times .876 \times 4.35$$

the computer will replace $92.7 \times .876$ by 81.2 and then multiply 81.2 by 4.35.

One additional unfortunate complication is due to design and engineering problems. Computers generally do not round the results of arithmetic computations. The last part of the execution of an arithmetic operation usually consists of ignoring those digits of the results which do not fit into a standard floating point word thereby contributing to the accumulation of errors. Let's look at some examples of how a 3-digit computer would do arithmetic if it chopped rather than rounded intermediate results.

Addition:

$$3.72 + 2.91 = 6.63 .$$

Since the answer fits into a 3-digit word, it appears correctly.

$$3.72 + .476 = 4.196$$

The computer result is 4.19.

$$14.6 + .0673 = 14.6673$$

The computer result is 14.6.

Subtraction:

$8.64 - 2.79 = 5.85$	Computer result: 5.85
$3.67 - 4.03 = -.36$	Computer result: -.360
$-18.3 - .0983 = -18.3983$	Computer result: -18.3
$1.23 - 1.22 = .01$	Computer result: .0100

Note the terminal zeros which have been inserted where necessary to achieve a 3-digit word.

Multiplication:

$4.27 \times 3.68 = 15.7136$ Computer result: 15.7

$27.3 \times .00364 = .099372$ Computer result: .0993

$.999 \times .999 = .998001$ Computer result: .998

Division:

$54.3 \div 4.55 = 11.934+$ Computer result: 11.9

$.0632 \div .00412 = 15.339+$ Computer result: 15.3

$27.5 \div .00987 = 2786.2+$ Computer result: 2780

Exercises 6-3

Do each problem using computer arithmetic as above (chopping all intermediate results to 3 digits) after first rounding all the given numbers to 3 digits where necessary.

1. Add: (a) $\begin{array}{r} 324.1 \\ 19.36 \\ \underline{124.08} \end{array}$ (b) $\begin{array}{r} 19.06 \\ 1.96 \\ \underline{25.0} \end{array}$

2. Subtract: (a) $\begin{array}{r} 8034 \\ \underline{-19.3} \end{array}$ (b) $\begin{array}{r} 27.601 \\ \underline{-3.4} \end{array}$ (c) $\begin{array}{r} 80.07 \\ \underline{-79.9} \end{array}$

3. Multiply: (a) $\begin{array}{r} .0037 \\ \underline{.0501} \end{array}$ (b) $\begin{array}{r} 2.06 \\ \underline{3.1} \end{array}$ (c) $\begin{array}{r} 12.6 \\ \underline{.0004} \end{array}$

4. Divide: (a) $227 \div 33$ (b) $1.9034 \div 1.5$ (c) $7.1 \div 1.0002$

5. Evaluate: (a) $19.03 + 1.007 - 10.3$

(b) $27.2 \times 1.3 - 1.8 \times 7.0$

(c) $\dfrac{101.1 - 3.1 \times 8.02}{14.105 + 1.9}$

6-4 Implications of Finite Word Length

The examples above are indicative of certain errors which may be introduced as a result of the execution of algorithms. In addition, there are surprising consequences of the fact that we have a fixed, finite word length for all numbers. Look at what this means. With our 3-digit computer, the number $\frac{1}{3}$ can best be represented as .333. Suppose we add $\frac{1}{3} + \frac{1}{3}$. The computer result is .666, and not .667 as it should be. In other words, when using a computer, the sum of the best representations of two numbers is not necessarily the best representation of the sum of the two numbers.

Another example of the same sort is $\frac{3}{16} + \frac{3}{16}$ which when performed in the 3-digit computer is .187 + .187 and yields .374 instead of .375, the best representation of $\frac{3}{8}$. One might think that if he took .188 as the best approximation to $\frac{3}{16}$, it would help but, alas, .188 + .188 = .376 and we are no closer to .375 than we were before.

Of course, if the word length were 8 digits, rather than 3, the inaccuracy would not be as great. In using computers, however, we often repeat calculations many times. Let's just see what happens if we add $\frac{1}{3}$ ten times successively, using 3-digit computer arithmetic.

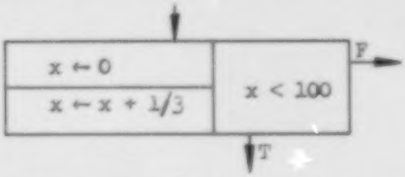
$$\begin{array}{r} .333 \\ +\ .333 \\ \hline .666 \\ +\ .333 \\ \hline .999 \\ +\ .333 \\ \hline 1.332 \end{array}$$

which becomes

$$\begin{array}{r} 1.33 \\ +\ \ .333 \\ \hline 1.66 \\ +\ \ .333 \\ \hline 1.99 \\ +\ \ .333 \\ \hline 2.32 \\ +\ \ .333 \\ \hline 2.65 \\ +\ \ .333 \\ \hline 2.98 \\ +\ \ .333 \\ \hline 3.31 \end{array}$$

As you can see, the error builds up quite rapidly. We could have obtained a better result by multiplying .333 by 10, getting 3.33.

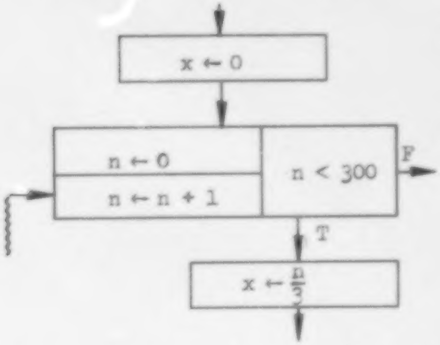
In Chapter 4 you learned how to devise algorithms for the construction of tables of values of a function. Suppose you had the task of constructing such a table of $f(x)$ for values of x from 0 to 100 at intervals of $\frac{1}{3}$. You might have incorporated a box like the one below into your flow chart.



With a three digit computer, the result of adding .333 300 times successively is 90.9, instead of the desired 100. Moreover, you would get to 100 after 331 additions. In other words, your table would have 331 lines rather than the desired 300. Interestingly enough, no matter how often you add .333 to 100 thereafter, the result would always be 100 in three-digit computer arithmetic. Thus, had your problem been the task of producing a table to 200 rather than 100, the three-digit computer would get caught in a loop and patiently print out identical consecutive lines with the value of x equal to 100 until someone caused the machine to stop.

If you had 8-digit word-length, the same thing would take a lot longer to happen, and things wouldn't be quite as bad.

Fortunately, as is usually the case, when the source of a problem has been identified, we can think of ways to improve things. Part of our trouble comes from repeated additions. We have seen above that multiplication reduces the error. So we can rewrite the box above as follows:



Another part of the problem arises from the fact that we commit an error as soon as we decide to use $\frac{1}{3}$ as the increment, since this number cannot be exactly represented in floating-point form. Might we do better if we used, say, .5 as an increment? First, we must determine whether the answers to this changed problem are in fact adequate. If this is the case, then indeed the errors which might have resulted from the solution of the original problem would not arise in this problem. Things are really not this easy, however. Many digital computers use the binary representation of numbers. It just so happens that the number $\frac{1}{2}$ has finite representations in both the decimal and binary systems of numeration, .5 and .1, respectively. The number $\frac{1}{10}$, however, does not have a finite binary representation. This easily overlooked fact could have a considerable effect on the accuracy of our results when our machine converts to binary for its actual computations.

To see that $\frac{1}{10}$ does not have a terminating representation in the binary system, we express 10 in binary form as $10 = 2^3 + 2^1 = 1010_{two}$ and divide 1 by it,

```
                .000110011001 ...
      1010 ⌐ 1.000000000000
                1010
                ----
                  1100
                  1010
                  ----
                    10000
                     1010
                    -----
                      1100
                      1010
                      ----
                        10000
                         1010
                        -----
```

We see that we are in a repeating cycle and that

$$\frac{1}{10} = .00011001100110011001100 \ldots \text{(base) two}$$

If our computer chops to 6 binary digits then

$$\frac{1}{10} = .000110011_{two}$$

in our computer arithmetic. Now if we compute

$$10 \cdot \frac{1}{10}$$

after converting to binary arithmetic using a 6 digit chop we have

$$\begin{array}{l} .000110011_{two} \\ \qquad\quad 1010_{two} \\ \hline .00110011 \\ .110011 \\ \hline .111111\boxed{11}_{two} \quad \text{CHOP} \end{array}$$

In our computer arithmetic we will then have

$$10 \cdot \frac{1}{10} = .111111_{two}$$

instead of

$$10 \cdot \frac{1}{10} = 1$$

The value of $.111111_{two}$ in more usual numeration is

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} = \frac{63}{64} .$$

Finally, if the computer outputs the answer in the decimal system it will convert $.111111_{two}$ to $.984375_{ten}$ which will be chopped to $.984_{ten}$.

In adding $\frac{1}{10}$ ten times with six binary digit chop after each addition the situation is much worse. Here we would have the binary result

$$.111101_{two} \text{ or } \frac{61}{64} .$$

This is .953125 in base ten which would be chopped to .953.

All of this shows us that we must know some details of how our computer works internally if we expect to use it effectively.

Now let us look at another kind of problem that arises out of the peculiarities of computer arithmetic. In Chapter 4 we discussed the evaluation of polynomial functions. This process occurs very frequently in a large variety of scientific problem areas. We told you earlier that the nested evaluation method for polynomials is preferable to computation of powers of x since it is more economical. Since fewer multiplications are required in the preferred method, you would expect greater accuracy, and, indeed, this is usually the case. But look at the following example.

Using three-digit computer arithmetic, it is required to evaluate the polynomial $x^3 - 6x^2 + 4x - .1$ for $x = 5.24$. For purposes of later discussion, imagine that this calculation is done as part of the larger problem of finding a root of the polynomial. In the table below we shall compute the exact value, without rounding, alongside the same calculations in three-digit computer arithmetic.

Exact Arithmetic	Computer Arithmetic
$x = 5.24$	$x = 5.24$
$x^2 = 27.4576$	$x^2 = 27.4$
$x^3 = 143.877824$	$x^3 = 143.$
$4x = 20.96$	$4x = 20.9$
$-6x^2 = -164.7456$	$-6x^2 = -164.$
$x^3 - 6x^2 = -20.867776$	$x^3 - 6x^2 = -21.0$
$x^3 - 6x^2 + 4x = .092224$	$x^3 - 6x^2 + 4x = -.100$
$x^3 - 6x^2 + 4x - .1 = -.007776$	$x^3 - 6x^2 + 4x - .1 = -.200$

Something would seem to be terribly wrong here. Of course, we are not using the recommended method, so let's try that one, using computer arithmetic, of course. Otherwise, we would get the same exact result as before. Here goes!

$$x - 6 = -.76$$

$$(x-6)x = x^2 - 6x = -3.98$$

$$(x-6)x + 4 = x^2 - 6x + 4 = .02$$

$$((x-6)x + 4)x = x^3 - 6x^2 + 4x = .104$$

$$x^3 - 6x^2 + 4x - .1 = .004$$

This doesn't look too good either. The last result, using the nested parentheses method is a lot closer than the previous method, but has the terrible disadvantage of having the wrong sign. This could cause havoc if we are trying to find a root of the polynomial, (see the discussion of bisection in Section 7-1) but the previous result which does have the correct sign is too much in error to be acceptable. This problem could be solved with reasonable accuracy on a real computer having, say, 8-digit words. But if you stop to consider that it is not uncommon to use computers to solve and evaluate polynomials of very high degree, the problem of accuracy is right back with us again. Unfortunately, no simple answer to this problem is known.

Exercises 6-4

1. Verify the value given in the text for adding $\frac{1}{10}$ ten times in a binary machine with a six binary digit chop.

2. Compute the binary values of $\frac{3}{10}$, $\frac{7}{10}$, $\frac{9}{10}$ by dividing and chopping to 6 binary digits.

3. How can the values of $\frac{2}{10}$, $\frac{4}{10}$, $\frac{6}{10}$ and $\frac{8}{10}$ (chopped to 6 binary digits) be obtained from previous results without using division? Find these values.

6-5 Non-Associativity of Computer Arithmetic

There are other problems and difficulties which arise in the process of doing arithmetic on computers. In the first method of evaluating the polynomial $x^3 - 6x^2 + 4x - .1$ for $x = 5.24$ we had computed the computer results for the first three terms as 143., 164., and 20.9, respectively. Evaluating $143. - 164. + 20.9 - .1$ from left to right, we get $-.200$ as the result. Since addition of real numbers is commutative and associative, we might have wanted to rearrange our work as follows:

$$(143. + 20.9) - (164. + .1) = 163. - 164. = -1.00 .$$

This is the most surprising result yet! Since the exact value of the polynomial is .007776, the result above is certainly the worst of the several approximations which we have obtained. But this result was obtained by simply changing the order of the addition, which does not affect the theoretical results. Comparing the intermediate results, it is of course easy to see that the difficulty lies in the chopping process after each operation.

Let's explore this phenomenon further.

Suppose we wish to compute

$$\sum_{n=1}^{10} \frac{1}{2^n} .$$

In the table below we have computed the exact decimal equivalents of the ten terms to be added, as well as their 3-digit computer equivalents. We have also added the exact values.

Computation of $\sum_{n=1}^{10} \frac{1}{2^n}$

n	$1/2^n$	Exact decimal equivalent	3-digit computer equivalent
1	1/2	.5	.500
2	1/4	.25	.250
3	1/8	.125	.125
4	1/16	.0625	.0625
5	1/32	.03125	.0312
6	1/64	.015625	.0156
7	1/128	.0078125	.00781
8	1/256	.00390625	.00390
9	1/512	.001953125	.00195
10	1/1024	.0009765625	.000976
		.9990234375	

Now let us add in the "normal" way, from top to bottom, using 3-digit computer arithmetic.

$.500 + .250 = .750$
$.750 + .125 = .875$
$.875 + .0625 = .937$
$.937 + .0312 = .968$
$.968 + .0156 = .983$
$.983 + .00781 = .990$
$.990 + .00390 = .993$
$.993 + .00195 = .994$
$.994 + .000976 = .994$

This result differs from the exact value by .005, which is not very good. Now let us try adding the same values in the reverse order.

$.000976 + .00195 = .00292$
$.00292 + .00390 = .00682$
$.00682 + .00781 = .0146$
$.0146 + .0156 = .0302$
$.0302 + .0312 = .0614$
$.0614 + .0625 = .123$
$.123 + .125 = .248$
$.248 + .250 = .498$
$.498 + .500 = .998$

Here the error from the exact result is only .001, or one-fifth of the previous error.

Again, it must be pointed out that we have been giving you examples with 3-digit arithmetic only so that you might be able to follow the step-by-step execution of the process with greater ease. Similar effects occur in real operations on real computers, where we perform much longer series of calculations. As an example, we have computed the sum of the first 10,000 terms of the series $\Sigma \frac{1}{n^2}$ by two algorithms.

First,

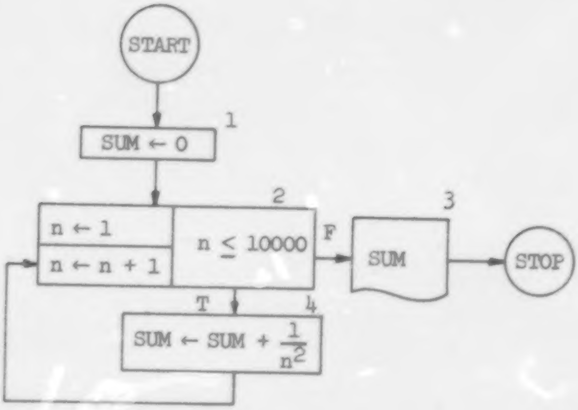

START
1
SUM ← 0
2
n ← 1
n ← n + 1
n ≤ 10000
F
3
SUM
STOP
T
4
SUM ← SUM + 1/n²

Figure 6-1. Summing a series forward

The result of executing this algorithm with 8-digit arithmetic is 1.6444743. Now, using the algorithm

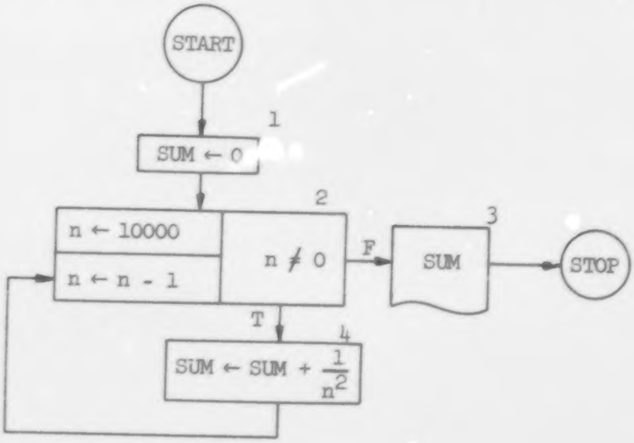

START
1
SUM ← 0
2
n ← 10000
n ← n - 1
n ≠ 0
F
3
SUM
STOP
T
4
SUM ← SUM + 1/n²

Figure 6-2. Summing a series backward

we get 1.6448339, a difference of .0003596.

Even if we modify these algorithms so as to add only 1000 terms of the series, the corresponding results are 1.6438868 and 1.6439344, differing by .0000476. Clearly, we seem to have established a valuable principle: The associative law does not work for computer arithmetic; therefore, the order in which computer operations are performed has a definite effect on the accuracy of the result.

In particular, it should be noted from the above examples that adding terms in order of increasing magnitude is distinctly preferable to the reverse order. By adding in the preferred way the cumulative effect of a large number of small terms has a better chance to make itself felt.

6-6 Some Pitfalls

For our next example, let us consider the problem of solving two simultaneous linear equations, i.e., a system[†]

$$a_{11}x + a_{12}y = b_1$$

$$a_{21}x + a_{22}y = b_2 .$$

In your earlier mathematics courses you have solved many such systems of equations, and you have learned several methods of doing so. Hopefully, you have also developed some shortcuts and tricks which simplify this job. These procedures depend on an examination of the system, some insight, and sometimes on a hunch.

For a digital-computer-oriented algorithm it is often possible, and even interesting, to attempt to develop a super-algorithm to examine the system, and, depending on the results of this examination, to choose one of several available sub-algorithms for the actual solution of the system. When you have finished studying this section, you may want to attempt the construction of such an algorithm. For the time being, we shall select a fixed method of solution in order to see what can go wrong.

Our method will consist of elimination of the variable x from the second equation by dividing all coefficients in the first equation by a_{11}, multiplying the resulting coefficients by a_{21}, and then subtracting the first equation from the second.

Let us first illustrate the algorithm by an example. To solve

(1) $2x + 3y = 12$

(2) $5x - 2y = 11$

we divide all coefficients of equation (1) by 2, obtaining

(3) $x + \frac{3}{2}y = 6$.

Next we multiply all coefficients of equation (3) by 5 and subtract the results from the corresponding coefficients of equation (2), which yields

---

[†] In customary mathematical usage, when no confusion is likely to result, the comma between double subscripts and/or the multiplication symbol between multiplicands may be omitted. From here on when these symbols obscure a pattern we wish to emphasize, we will sometimes omit the symbols.

(4) $-\frac{19}{2}y = -19.$

Dividing both sides of this equation by $-\frac{19}{2}$, we get

(5) $y = 2.$

We now have a new system of equations, (1) and (5), which is equivalent to the original system. Since the second component of the desired solution is the right-hand member of equation (5), we can substitute this value into equation (1) to get the first component, 3. Thus, the required solution is the ordered pair (3,2).

We now wish to describe this algorithm as a flow chart. In order to take care of contingencies we must check prior to each division that the quotient is not zero. Here is the flow chart for you to study.

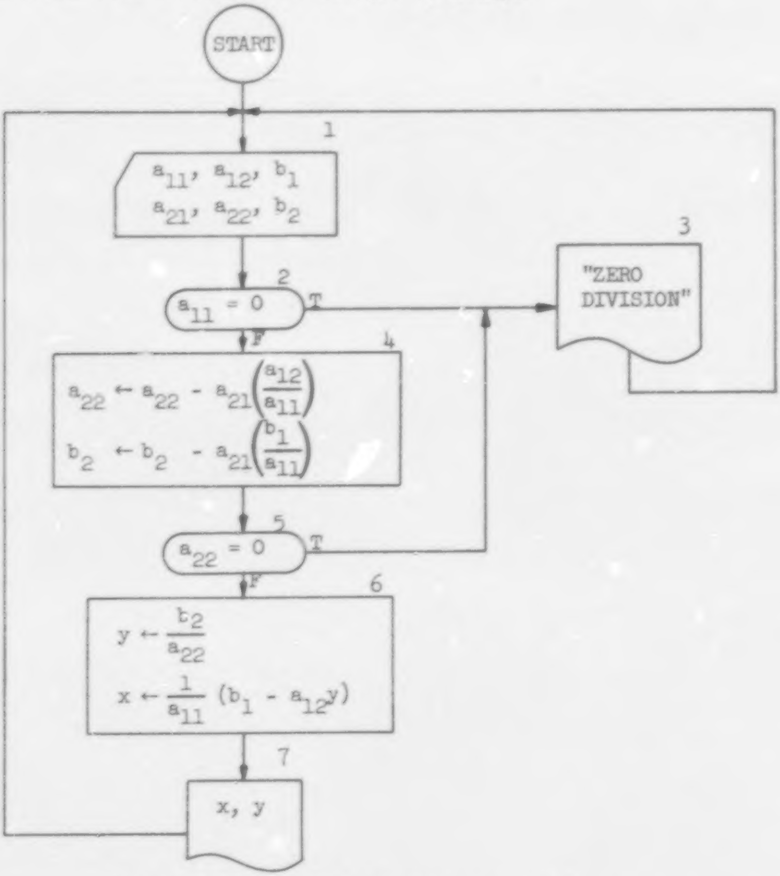

START
1
a_{11}, a_{12}, b_1
a_{21}, a_{22}, b_2
2
$a_{11} = 0$
T
F
3
"ZERO DIVISION"
4
$a_{22} \leftarrow a_{22} - a_{21}\left(\frac{a_{12}}{a_{11}}\right)$
$b_2 \leftarrow b_2 - a_{21}\left(\frac{b_1}{a_{11}}\right)$
5
$a_{22} = 0$
T
F
6
$y \leftarrow \frac{b_2}{a_{22}}$
$x \leftarrow \frac{1}{a_{11}}(b_1 - a_{12}y)$
7
x, y

Figure 6-3. Solving two linear equations

It would probably be helpful to you to hand-step through this flow chart with our earlier example, or one of your own, so as to convince you that this flow chart is indeed a description of the previously discussed algorithm, and, moreover, that the process does indeed produce the correct solution.

Now for a troublesome example. Consider the system

$$\begin{aligned} .0001x + y &= 1 \\ x + y &= 2. \end{aligned}$$

Let us first apply our algorithm using exact arithmetic to get the exact solution. a_{22} is to be replaced by $1 - 10000$, i.e., -9999, and b_2 becomes $2 - 10000$, i.e., -9998. Thus, the second equation becomes

$$-9999y = -9998$$

and the solution for y is $\frac{9998}{9999}$. By substitution we find the solution for x to be $\frac{10000}{9999}$. It is an easy matter to substitute these values into both equations and thus to verify the correctness of the solution.

Of course, neither of these values has an exact finite decimal representation, but, chopped to 10 significant digits, the solutions for x and y are 1.000100010 and .9998999899, respectively.

Now let us execute the algorithm again, but using 3-digit arithmetic. We again obtain the last equation above, but we must chop to 3 digits, which yields

$$-9990y = -9990$$

so that the solution for y is 1.00. Substituting this into the first equation, we get zero as the solution for x, which is a very bad result indeed.

What could have caused this extremely large error? To get at the trouble, let us repeat the execution of the algorithm once more, but with the order of the equations interchanged. In other words, let us solve the system

$$\begin{aligned} x + y &= 2 \\ .0001x + y &= 1 . \end{aligned}$$

Here we are to replace a_{22} by $1 - .0001$, which chopped to 3 digits is .999. Then we replace b_2 by $1 - .0002$, which chopped to 3 digits is also .999. So the solution for y is again 1.00, but substitution of this value into the first equation gives us 1.00 as the solution for x, which is not a bad result.

You are probably wondering why the order of the equations has such importance. The real problem is not the order of the equations, but the division by a_{11} in the algorithm which we have used. You will recall that the error in the division process was related to the magnitude of the divisor. In solving our system we had a choice of dividing by any one of the four coefficients which appear in the left-hand sides of the equations. We chose, with malice aforethought, to divide by the smallest coefficient we could find, thus maximizing the error. When the equations were interchanged, we divided by as large a coefficient as could be found, which gave us a good result.

In the next chapter you will find a more complete treatment of the important problem of solving simultaneous linear equations. Just to whet your appetite, you might be interested in knowing that there are many important and very real problems, such as the firing of guided missiles or the design of atomic reactors, which require the accurate solution of systems of thousands of simultaneous linear equations. So the problem is worth thinking about.

Lest you think that our explanation above has given you sufficient insight into the numerical aspects of the problem, we shall present another example of what can go wrong.

Consider the system

$$\begin{aligned} x + .98y &= 1.98 \\ .99x + .98y &= 1.97 \,. \end{aligned}$$

Using exact arithmetic, we get the solution $x = 1$, $y = 1$. Check it! Now if we change just one of the numbers involved in the problem slightly, say 1.97 to 1.96, we solve the system.

$$\begin{aligned} x + .98y &= 1.98 \\ .99x + .98y &= 1.96 \end{aligned}$$

we get the solution $x = 2$, $y = -\frac{2}{98} = -.0204$ to 3 significant digits. Thus, a very minor change in just one coefficient caused an extreme change in the solution.

There is a fairly simple explanation, of course, which involves trying to draw the graphs of the three equations involved in the two systems. We have not drawn these for you, as it would be very difficult to distinguish the three lines with the naked eye. Try to draw the graphs! You will see that all three are almost parallel. Since they are not parallel, however, the two pairs of lines have two distinct points of intersection, and you should be able to see why a very small change in a coefficient causes the result which we have observed. Try to construct similar examples of your own! Before leaving this subject we feel compelled to point out to you that when a real problem of this sort occurs, the coefficients in the system of equations to be solved are usually found by prior computations, which themselves are subject to error. This merely points out again why numerical analysis is such an important adjunct of computational procedure. It also should increase your respect for astronauts, as well as for the people who are responsible for the computational problems of a rocket launch.

6-7 More Pitfalls

Many numerical difficulties occur in computation as a result of using the number zero as a decision criterion. We have told you earlier that all real numbers are equally good. Nevertheless, some numbers are more important than others. You probably have observed in your earlier studies in mathematics that the number zero occurs more frequently than any other.

In abstract mathematics you may have realized that there does not exist a smallest positive number. To prove this, it is sufficient to observe that if anyone claimed to know such a number, one-half of that number would also be positive, but smaller. In other words, we can find positive numbers arbitrarily close to zero.

In computer mathematics, this is not the case. For each specific computer system there exists a specific smallest positive number. Therefore, many mathematical ideas, theorems, and algorithms, whose abstract justification depends on being able to find arbitrarily small positive numbers, must be modified for computer use. While a complete treatment of this difference is far beyond the scope of this book, we can illustrate some consequences of the differences by means of some fairly simple examples.

You have already considered some of the logical problems involved in devising an algorithm for the solution of an equation of the form $ax^2 + bx + c = 0$, given the coefficients a, b, and c. The logical problems included checking to see if a was zero, examining the discriminant $b^2 - 4ac$ to see if it was negative, zero or positive, etc. So you see that the number zero plays an important part in this logical analysis. If a is zero, the equation is not quadratic, and therefore cannot have exactly two roots. If $a \neq 0$, the equation is quadratic and has exactly two roots. Suppose further that the discriminant is positive, so that the two roots are real. You then know how to find the roots by the quadratic formula.

Suppose we wish to solve the equation $x^2 - 6x + 4 = 0$, which satisfies the criteria above. The exact roots are $3 \pm \sqrt{5}$. These roots are irrational, however. Therefore, no number representable digitally in a computer can be a solution. In other words, if we asked the question: Is $x^2 - 6x + 4 = 0$?; then the answer as given by a computer would always be: NO. We can, however, determine that the value of the left side of the equation is $-.0271$ when x is 5.23, and is $.0176$ when x is 5.24. Therefore, there is a root between 5.23 and 5.24, and we could choose one of these as an approximation to the root. We could also compute closer approximations. The important thing to bear in mind is not to ask for an exact solution, but for an approximation with a specified approximation criterion. Such a criterion might be the value of the left side of the equation. The choice of such a value is not easy, however, and requires considerable analysis for proper determination.

As another example, consider the equation

$$x^2 + 10000x - 1 = 0 .$$

The discriminant, $b^2 - 4ac = 100000004$, and $\sqrt{b^2 - 4ac} = 10000.00020$, correct to 10 digits. We can compute the roots as -10000.0001 and .0001. Substituting either of these values, the left side of the equation becomes 10^{-8}, which is acceptable. Supposing we can only use 8-digit arithmetic, however, then $\sqrt{b^2 - 4ac} = b$, numerically, and we get -10000 and 0 as roots. While the inaccuracy thus introduced into the computations is relatively small, it is aesthetically disturbing to get zero as a result, while the equation does in fact have a positive root. This difficulty can be avoided by remembering that if $a \neq 0$, the product of the roots of the equation is $\frac{c}{a}$. We can therefore get the positive root accurately even with 4-digit arithmetic by dividing -1 by -10000. So, particularly when b^2 is much larger than $|4ac|$ we can enhance the accuracy of computation of the root near zero by dividing $\frac{c}{a}$ by the numerically large root. This avoids getting zero as the result of subtracting the nearly equal quantities b and $\sqrt{b^2 - 4ac}$.

6-8 Approximating Functions

You have had some experience in using mathematical tables in solving problems. Specifically, you probably have used tables of logarithms, trigonometric functions, and logarithms of trigonometric functions. You may not have given much thought to the origin of these tables, to the methods by which they were created. The mathematical theories used in this process are too involved to be detailed here. On the other hand, the storage capacities of digital computers are limited, and preclude the storage of all the tables which users might need for the solution of their variegated problems. If you have not already done so, you might enjoy looking at the Handbook of Mathematical Tables published by the Chemical Rubber Company, just to get an idea of the multiplicity of tables in existence. There are many more books of tables in existence. Clearly we could not begin to store all of this information in a computer. So to do the necessary calculations we incorporate subroutines into programs which calculate functional values. In hand calculations these values are usually obtained by looking into books of tables. We will conclude this chapter with a brief discussion of a few methods of computing values of some of the most common such functions. These are usually included as library functions with most computing systems.

The most common function is the square root function. There are several methods of computing square roots. The most common of these you met in Chapter 5. Suppose we use the Newton Method flow chart of Figure 5-7 repeated here for your convenience as Figure 6-4, to compute the square root of 2. We present the successive results in tabular form.

Table 6-1

Newton's Square Root Method

g	a/g	h	$\|h - g\|$
1	$2/1 = 2$	$\frac{1}{2}(1+2) = 1.5$	.5
1.5	$2/1.5 = 1.3333$	$\frac{1}{2}(1.5+1.3333) = 1.4166$	.0834
1.4166	$2/1.4166 = 1.4118$	$\frac{1}{2}(1.4166+1.4118) = 1.4142$	.0024
1.4142	$2/1.4142 = 1.4142$	$\frac{1}{2}(1.4142+1.4142) = 1.4142$	.0000

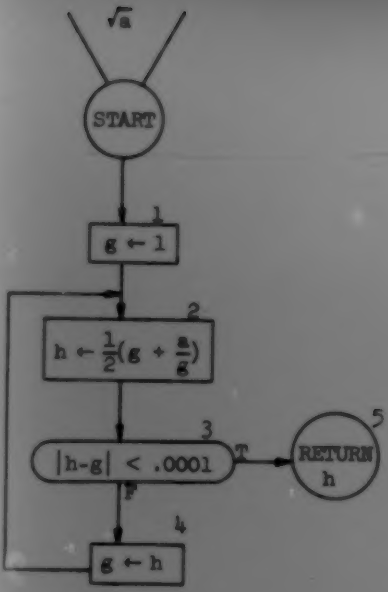


Figure 6-4. Newton square root method

The computation shown in Table 6-1 is terminated after four iterations because further iteration would not change the result. This is not necessarily a good criterion however, since it may not always be possible to achieve and, moreover, does not necessarily give the desired result. Note that $(1.414)^2 = 1.999396$, but it actually is the best 4-digit approximation since $(1.415)^2 = 2.002225$.

Suppose we desired a 3-digit approximation to $\sqrt{.709}$ and somehow we arrived at .840 as an approximate result. Note that, using 3-digit arithmetic,

$$.709 \div .840 = .844$$

$$\frac{1}{2}(.840 + .844) = \frac{1}{2}(1.68) = .840 .$$

Thus, although .842 is a better 3-digit approximation to $\sqrt{.709}$, the "averaging" algorithm described will yield .840 if three-digit arithmetic is used. The moral is that it is not desirable to require accuracy to the word-length of the machine. The algorithm should be terminated whenever two consecutive approximations differ by less than a specified maximum allowable error, which should be significantly greater than the smallest positive number representable in the machine.

Finally, we shall briefly look at computation of the sine function. When you first learned about this function you were told that values of this function typically are irrational numbers, which could be computed to any desired degree of accuracy by methods developed in calculus. Since you probably have not yet studied much of calculus, it is not possible to go into these methods at this point. We can, however, show you examples of how values of the sine function can be computed.

We are considering the function $\sin x$, where x is a real number, which you can think of as the radian measure of an angle if you so desire. It is proved in calculus that the sum of "sufficiently many" terms of the series

$$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

is a number close to $\sin x$, and that the difference between $\sin x$ and the abovementioned sum becomes numerically smaller as more and more terms are taken.

If we look at the series from a computational view, the calculations are not very difficult. Let's look at an easy example, say the calculation of $\sin(.3)$.

The first term is, of course, the easiest.

To get the numerator of the second term, we multiply $.3$ by the value of x^2, in this case $.09$. To get the denominator, we multiply the denominator of the previous term, 1, by $2 \cdot 3$.

To get the numerator of the third term, we multiply the numerator of the previous term again by $.09$, and to get the denominator, we multiply the previous denominator by $4 \cdot 5$.

This seems like a fairly simple procedure, so let us compute the first few terms. To simplify things, let us name the terms as components of a vector, $y_1, y_2, \ldots$.

$$y_1 = x = .3$$

$$y_2 = \frac{-x^3}{3!} = \frac{-x^2}{2 \cdot 3} \cdot y_1 = \frac{(-.09) \cdot (.3)}{6} = -.0045$$

$$y_3 = \frac{x^5}{5!} = -\frac{x^2}{4 \cdot 5} \cdot y_2 = \frac{(-.09) \cdot (-.0045)}{20} = .00002025$$

$$y_4 = \frac{-x^7}{7!} = \frac{-x^2}{6 \cdot 7} \cdot y_3 = \frac{(-.09) \cdot (.00002025)}{42} = -.00000004339$$

Clearly, subsequent terms will be numerically much smaller than those which we have computed. If we stop at this point and compute the sum of these four terms, we obtain .2955202066 as an approximation to sin(.3). Published 5-place tables give .29552, so that we have perfect agreement to five digits.

Now let's try to compute sin 5 correct to 5 digits.

$$y_1 = x = 5.$$

$$y_2 = \frac{-x^2}{2\cdot 3}\cdot y_1 = \frac{-25\cdot 5}{6} = -20.0416666$$

$$y_3 = \frac{-x^2}{4\cdot 5}\cdot y_2 = \frac{(-25)\cdot(-20.8333333)}{20} = -26.0416666$$

$$y_4 = \frac{-x^2}{6\cdot 7}\cdot y_3 = \frac{(-25)\cdot(26.0416666)}{42} = -15.5009920$$

$$y_5 = \frac{-x^2}{8\cdot 9}\cdot y_4 = \frac{(-25)\cdot(-15.5009920)}{72} = 5.3822889$$

$$y_6 = \frac{-x^2}{10\cdot 11}\cdot y_5 = \frac{(-25)\cdot(5.3822889)}{110} = -1.2232475$$

$$y_7 = \frac{-x^2}{12\cdot 13}\cdot y_6 = \frac{(-25)\cdot(-1.2232475)}{156} = .1960333$$

$$y_8 = \frac{-x^2}{14\cdot 15}\cdot y_7 = \frac{(-25)\cdot(.1960333)}{210} = -.0233373$$

$$y_9 = \frac{-x^2}{16\cdot 17}\cdot y_8 = \frac{(-25)\cdot(-.0233373)}{272} = .0021450$$

$$y_{10} = \frac{-x^2}{18\cdot 19}\cdot y_9 = \frac{(-25)\cdot(.0021450)}{342} = -.0001568$$

$$y_{11} = \frac{-x^2}{20\cdot 21}\cdot y_{10} = \frac{(-25)\cdot(-.0001568)}{420} = .0000093$$

$$y_{12} = \frac{-x^2}{22\cdot 23}\cdot y_{11} = \frac{(-25)\cdot(.0000093)}{506} = -.0000005$$

Note that a larger number of terms was necessary to achieve the required accuracy for this larger value of x. We have arbitrarily kept 7 digits to the right of the decimal point, which might not always be possible with a digital computer.

Adding our 12 terms, we obtain -.9589243 as the approximation to sin 5, and we can feel fairly confident of the accuracy of the first five digits.

We give in Figure 6-5 a flow chart for computing the value of sin(x).

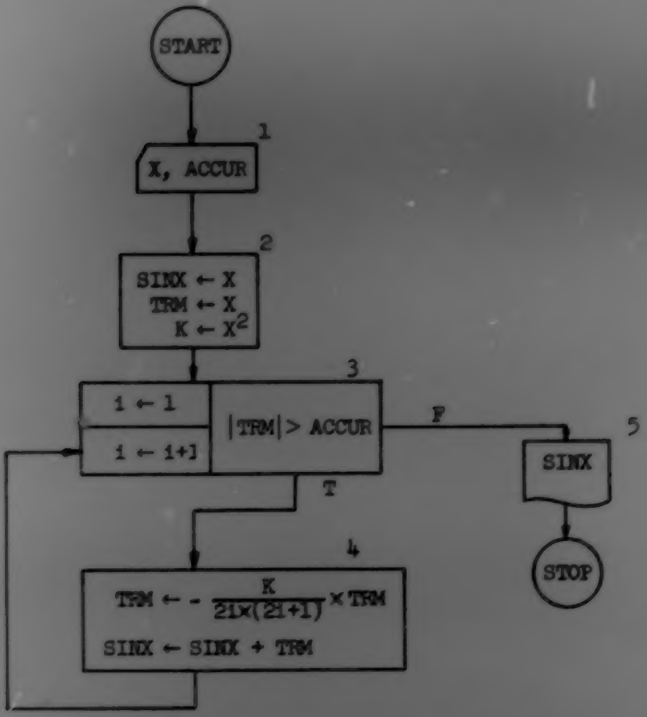

START
1
X, ACCUR
2
SINX ← X
TRM ← X
K ← X²
3
i ← 1
i ← i+1
|TRM| > ACCUR
F
T
5
SINX
STOP
4
TRM ← - K/(2i×(2i+1)) × TRM
SINX ← SINX + TRM

Figure 6-5. Algorithm to compute sin(x)

Note that we do not store the individual terms of the series. Instead, we sum them as they are computed, keeping a copy only of the very last term (TRM). In this way it is possible to avoid the use of subscripts.

We have seen in the preceding examples that when $|x|$ is large the computation in Figure 6-5 is cumbersome and subject to many numerical inaccuracies. Therefore it is desirable to convert the problem to calculation of an approximation for small values of x by using the reduction formulas for trigonometric functions which you have studied in an earlier course. Accordingly you will be asked in problem 4 below to draw a modification of Figure 6-5 incorporating this conversion.

Exercises 6-8

1. What would be the result of applying the square root algorithm of Figure 6-4 if the input variable a has a value 0 ?

2. What would be the result of applying the square root algorithm of Figure 6-4 if the input, a, is less than 0 ?

3. The square root algorithm, Figure 6-4, uses $g = 1$ as an initial approximation no matter what value a has.

   (a) What other initial approximations might be tried?

   (b) Suggest what advantages or disadvantages such approximations might have.

4. Modify the algorithm which computes the sin(x), Figure 6-5, so that the value of x is reduced to one between $\frac{-\pi}{2}$ and $\frac{\pi}{2}$ using the relation $\sin x = (-1)^n \sin(x - n\pi)$, for any integer, n.

Chapter 7

SOME MATHEMATICAL APPLICATIONS

7-1 Root of an Equation by Bisection

Locating a Root by Graphing

The problem of finding the roots of equations is a very common one. Very early in your study of algebra you learned how to solve a linear equation and a little bit later you learned how to solve a quadratic equation. Also, the solution of systems of linear equations is familiar to you. Indeed, we will study this problem in Sections 7-4 and 7-5. But suppose we have an equation of higher degree than a quadratic or an equation that involves some functions like $\sin x$, $\cos x$, or $\tan x$ besides the ordinary algebraic functions of x. For example, we might want the roots of the equation

$$3x^3 - 7x - 2 = 0$$

or the equation

$$x^5 - 4x^4 + 7x^3 - x + 3 = 0.^{\dagger}$$

Or we might want to solve the equation

$$x + \ell n x = 0$$

or the equation

$$x = \tan x.$$

There are many methods which have been proposed for finding the roots of such equations. Perhaps one of the simplest is a graphical method. If the equation is written in the form $f(x) = 0$, then we have only to calculate $f(x)$ for a suitable set of values of x and plot the graph of $y = f(x)$. Whenever this graph crosses the x-axis there will be a root of the equation. Of course, we can get only an approximate result by such a graphical procedure because of the limitations on our ability to draw a graph very accurately. We also may have difficulty in finding the right domain of the values of x to use in plotting the graph.

[†] In customary mathematical usage, when no confusion is likely to result, the comma between double subscripts and/or the multiplication symbol between multiplicands may be omitted. Henceforth in this work when the visual distraction of these symbols obscures a pattern we wish to emphasize, we shall omit the symbols. Also, parentheses around function arguments are sometimes omitted in the text but never in flow charts.

Suppose that we want to find approximately the roots of the equation

$$3x^3 - 7x - 2 = 0.$$

We draw the graph of $y = 3x^3 - 7x - 2$. To do this we first calculate a small table of values.

x	-3	-2	-1	0	1	2	3
y	-62	-12	2	-2	-6	8	58

The graph is shown in Figure 7-1.

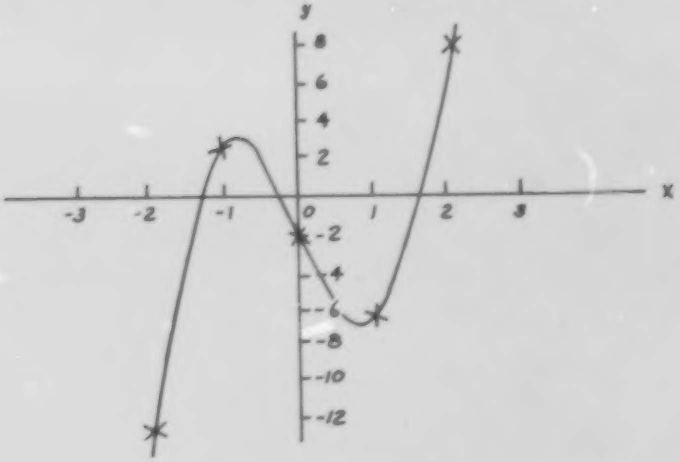


Figure 7-1. Graph of $y = 3x^3 - 7x - 2$

The roots are seen to be near -1, between -1 and 0 and between 1 and 2.

An alternative method is to write the equation to be solved in the form $f_1(x) = f_2(x)$ and plot the graphs of $y = f_1(x)$ and $y = f_2(x)$. The x-coordinates of the points where these graphs intersect will give us the roots of the equation.

For example, suppose we want to find approximately the roots of the equation $x = \cot x$. We draw the graphs of $y = x$ and $y = \cot x$. The x-coordinates of the points where these curves intersect give us the roots of the equation.

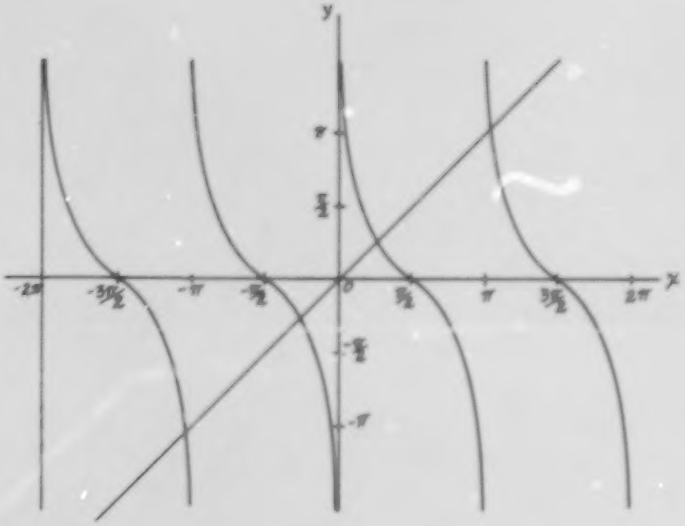


Figure 7-2. Graphs of $y = x$ and $y = \cot x$.

We see that there are infinitely many roots, the smallest ones being near $\pm \pi/4$. The other roots lie near to $\pm k\pi$, $k = 1,2,3, \ldots$.

Exercises 7-1 Set A

1. By plotting the graph of $y = f(x)$ where $f(x)$ denotes the lefthand side in each of the following five equations, find approximately the roots of the following equations.

   (a) $x^3 - 2x - 5 = 0$

   (b) $x^4 + 3x^2 - 2x - 4 = 0$

   (c) $3x^4 - 2x^3 + 7x - 4 = 0$

   (d) $x^3 - x - 1 = 0$

   (e) $x^2 - 3x - 4 \sin^2 x = 0$

2. By plotting the graphs of $y = f_1(x)$ and $y = f_2(x)$ for suitably chosen $f_1(x)$ and $f_2(x)$ find approximately the roots of the following equations.

   (a) $x = \tan x$

   (b) $x + \ln x = 0$

   (c) $5 - x = 5 \sin x$

The Method of Successive Bisection

Either of these graphical methods will give us an approximation to a root of the equation. Once we have an idea of where the root lies we can improve the accuracy of the root.

One of the most powerful methods but often not the most efficient method for finding a root of an equation in a given interval is the method of successive bisection. The method is designed for use when the function is known in advance to be continuous (i.e., no breaks in the graph) and to have just one root in the given interval. We consider it incidental that the method will also produce one of the roots in the case that the function has an odd number of roots in the interval. If the number of roots in the interval is even, the method is inapplicable.

Suppose that we seek a root of the equation $f(x) = 0$ where $f(x)$ is a function of x. Suppose further that $f(x_1) < 0$ and $f(x_2) > 0$, i.e., the graph of $y = f(x)$ is below the x-axis at $x = x_1$ and above the x-axis at $x = x_2$. This situation is illustrated in Figure 7-3. Now if the graph of $y = f(x)$ has no gaps or jumps between $x = x_1$ and $x = x_2$, then it must cross the x-axis between x_1 and x_2 and, hence, there must be a root of $f(x) = 0$ between x_1 and x_2.

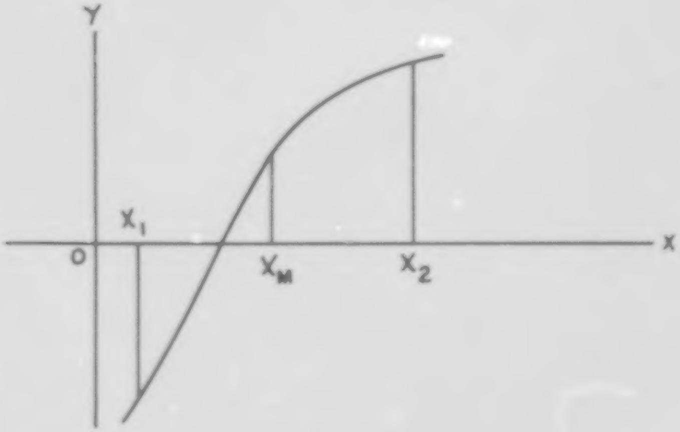


Figure 7-3. Graph of some $f(x)$ with a root between x_1 and x_2

We now bisect the interval (x_1,x_2) and denote the midpoint by x_M so that we have

$$x_M = (x_1 + x_2)/2.$$

If $f(x_M) = 0$, then we have a root. However, if $f(x_M) > 0$, as in Figure 7-3, then there is a root between x_1 and x_M. So to prepare for the next step, we assign the value of x_M to the variable x_2. Thus, again we can denote the interval in which the root lies by (x_1,x_2), but the length of our new interval is half that of the original interval, as shown in Figure 7-3a.

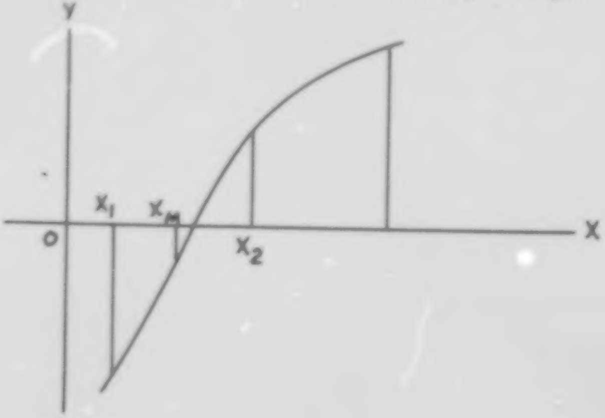


Figure 7-3a. New interval after one bisection

We calculate the value of the function at the midpoint x_M of the new interval. This time $f(x_M) < 0$, as shown in Figure 7-3a, and therefore the root is between x_M and x_2. Again we have isolated the root in an interval half the length of the previous interval. If we now assign the value of x_M to the variable x_1, we can again denote the current interval containing the root by (x_1,x_2).

We may repeat the bisection process for the new interval in which we know the root lies. By repeating this bisection process we can come as close to the root as we please for at each step we halve the length of the interval in which the root lies. Thus, 10 steps will reduce the length of the interval by a factor of 2^{10} or roughly 1000 while 20 steps will reduce it by a factor of 2^{20} or roughly 1000000. Thus, the method is seen to be moderately effective.

Example:

Now let us consider again the equation

$$3x^3 - 7x - 2 = 0.$$

The corresponding graph is drawn in Figure 7-1. If we let $f(x) = 3x^3 - 7x - 2$, we see that

$$f(1) < 0 \quad \text{and} \quad f(2) > 0$$

and so we know there is a root of the equation between 1 and 2.

We now bisect this interval. The midpoint is $x_M = 3/2$. We easily find by substitution

$$f(\tfrac{3}{2}) = -19/8 \quad \text{so} \quad f(\tfrac{3}{2}) < 0.$$

Thus, the root lies in the interval $(3/2,2)$. The midpoint of this interval is $x_M = 7/4$. But

$$f(\tfrac{7}{4}) = 117/64 \quad \text{so} \quad f(\tfrac{7}{4}) > 0.$$

Hence, the root lies in the interval $(3/2,7/4)$.

We can continue this process as many times as we wish, each time finding in which interval the root lies. Note that the length of the interval is halved at each step.

Exercises 7-1 Set B

Use the method of bisection to find approximate values of the indicated roots of the following equations. In each case start with the indicated interval which is known to contain a root and use the indicated number of bisection steps.

1.	$x^3 - 2x - 5 = 0,$	$(2,3)$	4 steps
2.	$x^4 + 3x^2 - 2x - 4 = 0,$	$(-1,0)$	3 steps
3.	$x = \tan x,$	$(3,5)$	4 steps

Compare your results with the results found graphically in Exercises 7-1A.

Now we will develop a flow chart based on the successive bisection method for approximating a root (or "zero") of a given function F in a given interval. In passing to the flow chart stage we will use xl, x2 and xM instead of x_1, x_2 and x_M.

The basic operation of the bisection process is the replacement of the interval (xl,x2) in which a root of F is known to lie, by a subinterval of half its length in which the root is known to lie. If we assume that the initial values of xl and x2 are such that F(xl) and F(x2) have opposite signs, then the partial flow chart of Figure 7-4 describes the steps of this operation.

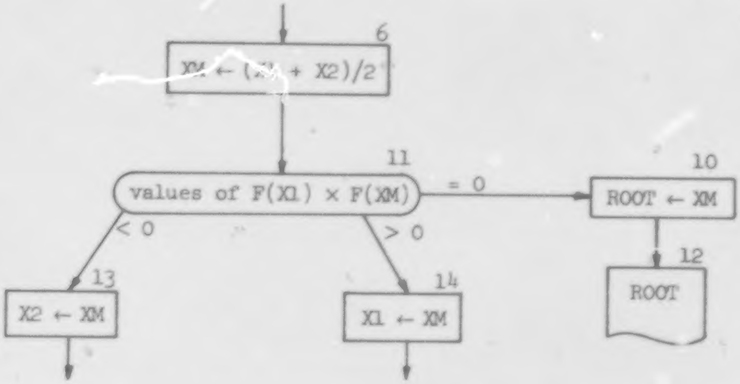


Figure 7-4. Partial flow chart of bisection procedure

In box 6 the midpoint XM is calculated. In box 11 we see the easiest way of deciding whether F(X1) and F(XM) have the same or opposite signs. They have the same or opposite signs according to whether their product is positive or negative. If their product is zero, then F(XM) must be zero as we are assuming that F(X1) is already known to be different from zero.

At each stage, before we decide to replace the interval (X1,X2) by an interval half as long, we need to check the length of the interval, i.e., the absolute value of the difference, X1 - X2. If it is sufficiently small, (say smaller than a given value ϵ), we accept the value XM at the midpoint of the interval as the root of the equation. Otherwise, we repeat the operations of the flow chart of Figure 7-4.

We are now ready to draw a complete flow chart for finding a root by the bisection method. It is seen in Figure 7-5. We assume the equation is given in the form F(X) = 0. We are given two numbers, the initial values of X1 and X2 between which a root is supposed to lie. We also assume that a tolerance ϵ is given and that we are supposed to calculate the root with an error less than ϵ. More precisely, the true root is to lie within an interval of length ϵ centered on the calculated root. Values of X1, X2 and ϵ are read in box 1 of Figure 7-5.

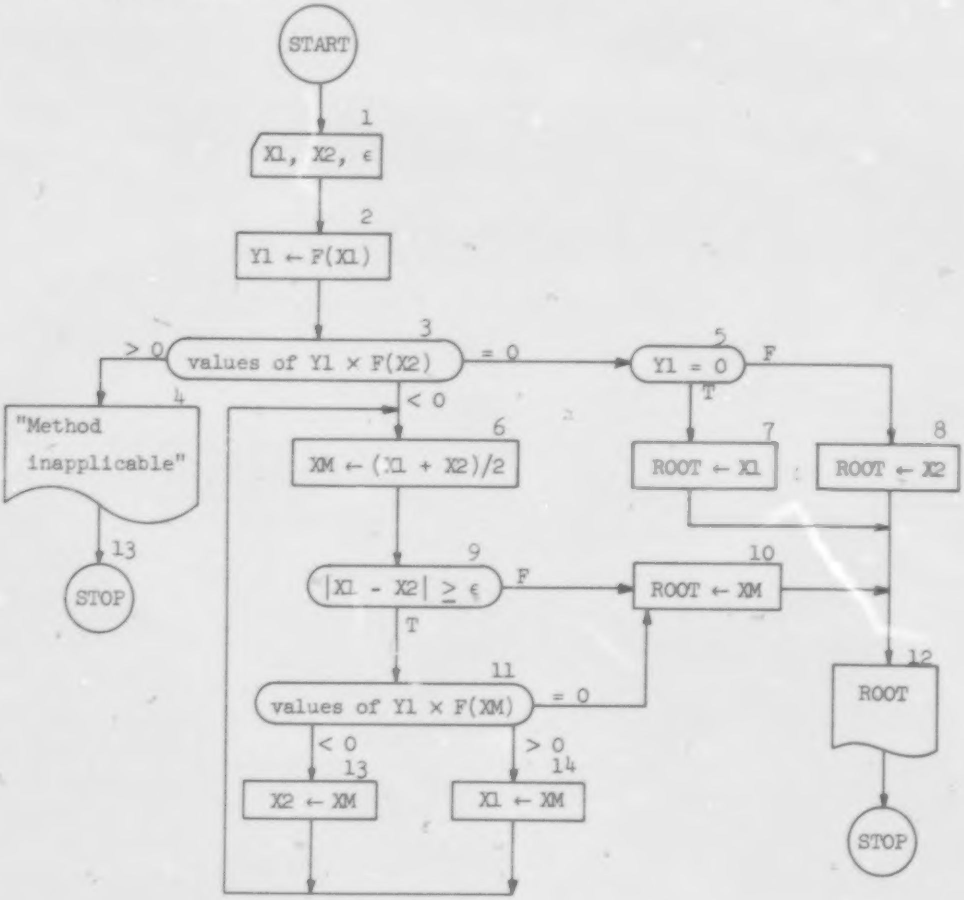

START
1
X1, X2, ϵ
2
Y1 ← F(X1)
3
values of Y1 × F(X2)
> 0
= 0
< 0
4
"Method inapplicable"
13
STOP
5
Y1 = 0
F
T
6
XM ← (X1 + X2)/2
7
ROOT ← X1
8
ROOT ← X2
9
|X1 - X2| ≥ ϵ
F
T
10
ROOT ← XM
11
values of Y1 × F(XM)
= 0
< 0
> 0
13
X2 ← XM
14
X1 ← XM
12
ROOT
STOP

Figure 7-5. Bisect flow chart

In box 2 the value of F(X1) is assigned to an auxiliary variable Y1. This is based on the principal that if you are going to use a particular value of an expression several times, you assign that value to some variable to avoid repeating the identical computation. The fragment of Figure 7-4 is repeated in boxes 6, 10, 11, 12, 13 and 14 of Figure 7-5. Box 3 is a test to determine whether the initial values of F(X1) and F(X2) have the same or opposite signs. If they have opposite signs, then $Y1 \cdot F(X1)$ is negative, so we go to box 6 and start the bisection process. If F(X1) and F(X2) have opposite signs, then $Y1 \cdot F(X2)$ is negative and the method is inapplicable. This is indicated in box 4. If $Y1 \cdot F(X2) = 0$, then either X1 or X2 is already a root and the purpose of boxes 5, 7 and 8 is to find out which. Note that if both X1 and X2 are roots, we will not discover this but will be satisfied with the one root X1.

One little trick remains to be explained. We note that only one assignment has been made to the variable Y1 so that its value never changes; its value is always F at the initial value of X1. We want, in box 11, to determine whether F(X1) and F(XM) have the same or different signs. Only if the signs are the same do we assign XM to X1. But then the new F(X1) and the old one will have the same sign. In other words, the sign of F(X1) never changes. The test of sign in box 11 can thus as well be made using the initial value of F(X1) as with the latest value.

Exercises 7-1 Set C

Step through the flow chart of Figure 7-5 with the indicated functions, the indicated intervals and the indicated values of ϵ. Determine whether there are an odd number of roots in the interval and, if there are, determine the value of ROOT. Sliderule accuracy is adequate. Tables may be used in Problems 2 and 3.

1. $x^3 - x - 1 = 0$ $[0,2]$ $\epsilon = 0.1$
2. $x + \ln x = 0$ $[.1,1]$ $\epsilon = 0.15$
3. $5 - x = 5 \sin x$ $[0,2]$ $\epsilon = 0.4$
4. $x^3 - 3x - 2 = 0$ $[0,2]$ $\epsilon = 0.1$
5. $x^3 - 2x^2 - 13x - 10 = 0$ $[0,4]$ $\epsilon = 0.1$

The following example should be used as a guide:

Problem:

$$3x^4 - 2x^3 + 7x - 4 = 0 \qquad [0,1] \qquad \epsilon = 0.4$$

Solution:

An odd number of roots.

For $\epsilon = 0.4$ the root is 0.625. $f(x) = 3x^4 - 2x^3 + 7x - 4 = 0$

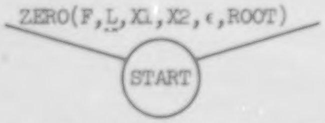
Step	x_1	Sign of $f(x_1)$	x_2	Sign of $f(x_2)$	x_m	Sign of $f(x_m)$	$\lvert x_1 - x_2 \rvert$
	0	-	1	+	0.5	-	1
1	0.5	-	1	+	0.75	+	0.5
2	0.5	-	0.75	+	0.625	+	0.25

Bisection as a Procedure

After you have traced through the details of Bisect in Figure 7-5, and made sure it is in good working order, we are ready to proceed with converting it into a procedure. Bisect always operates on a function F.

We will let one of the parameters be a dummy function name. In this way the procedure could solve for the roots of functions having a variety of actual names. If we don't add a dummy function name (F) to the parameter list, we could not search for a root of G(X) or R(X), for example, and we would be unable to have one main flow chart call on the procedure and refer to more than one function. Clearly, a function name should be an element of the parameter list when we convert Bisect to a procedure.

Since Figure 7-5 has an alternate exit, it also seems natural to include a label (box number) in the parameter list. Calling the bisect procedure ZERO, we show the funnel as it would replace the START and box 1 of Figure 7-5.



Here the F is a placeholder for a function reference that will be supplied by the main flow chart. The L provides a location for a statement label as described in Section 5-5.

Figure 7-6 shows the complete flow chart for our new procedure ZERO.

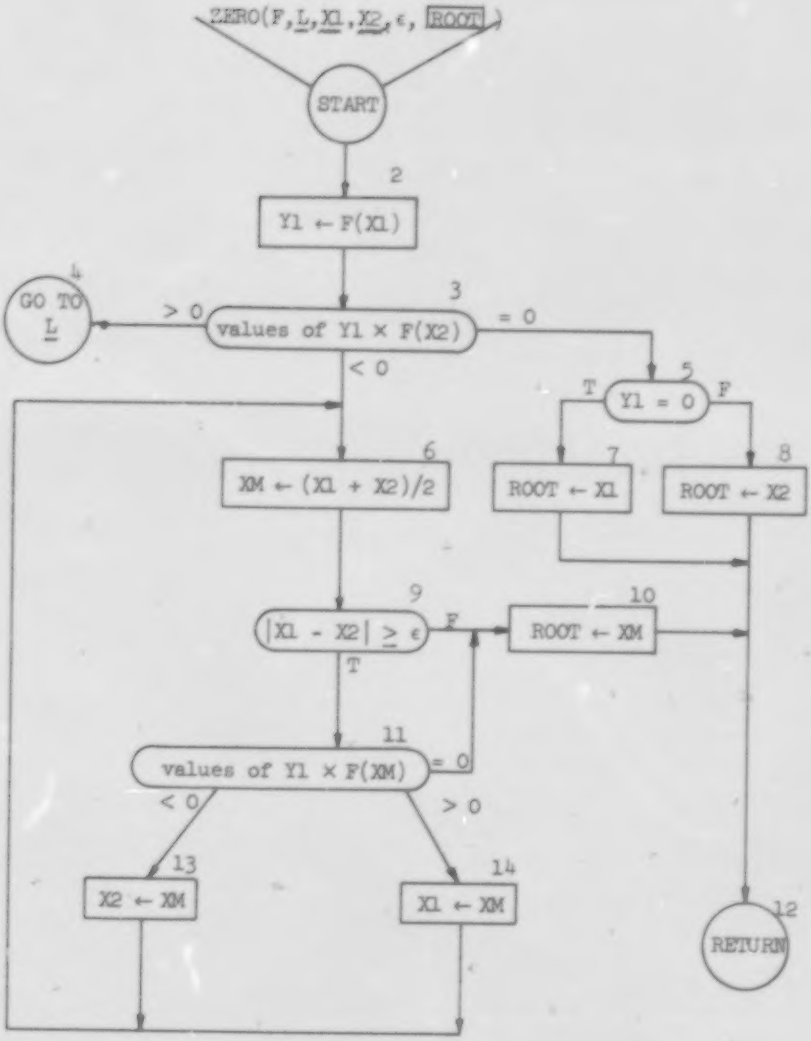

ZERO(F,L,X1,X2,ε,ROOT)
START
2
Y1 ← F(X1)
3
values of Y1 × F(X2)
> 0
4
GO TO L
= 0
< 0
5
Y1 = 0
T
F
6
XM ← (X1 + X2)/2
7
ROOT ← X1
8
ROOT ← X2
9
|X1 - X2| ≥ ε
F
T
10
ROOT ← XM
11
values of Y1 × F(XM)
= 0
< 0
> 0
13
X2 ← XM
14
X1 ← XM
12
RETURN

Figure 7-6. ZERO: bisection procedure

We notice some fancy new ornamentation in the funnel of Figure 7-6. To understand it, recall from Section 5-5 that variables in the funnel of a procedure can be thought of as either slips of paper or window boxes coming into the hopper of the brick chamber. In this procedure some variables are treated in one of these ways and some in the other.

Let us concentrate our attention on the variables X1 and X2 in the funnel. X1 and X2 start as the ends of our original interval. Recall that our procedure has as its only purpose the locating of a root. The procedure should not make any capricious alterations in the values of the main flow chart variables. Looking at boxes 13 and 14 we see that assignment is made to X1 and X2. Therefore, if the values of X1 and X2 are brought to the subroutine in their window boxes, these boxes will be returned to the main flow chart with changed values. This will throw a monkey wrench into the main flow chart if original values of these variables are needed later on. We insist, therefore, that the initial values to be assigned to X1 and X2 should be brought to the hopper on slips of paper. To remind ourselves of this fact we have adopted the convention of underlining these variables with wavy lines.

We are indifferent as to which of the two treatments to use on the variable ϵ as its value is unchanged in the course of the procedure. This is reflected in our not giving ϵ any special marking. [Our preference is for having its value transmitted by assignment.]

Now, finally we look at the dummy "output" variable, ROOT. Remember that the sole purpose of this variable is to hold the information going back to the main flow chart. Clearly, a receptacle must be provided to carry this information back. The window box belonging to the main flow chart variable will therefore come through the hopper and be relabeled as ROOT. We have enclosed this variable in a rectangle to keep track of this fact.

We will adhere to these rectangles and wavy line conventions when we wish to emphasize the way in which the variables must be treated. This should be a help to you in writing your procedural language programs. Without this convention you might be forced to search through the reference flow chart if you have forgotten what the variables are doing.

Before leaving this subject we want to point out that the danger of mistreating variables X1 and X2 could have been avoided in another way. The top of the flow chart of Figure 7-6 could have been altered to use X1 and X2 as auxiliary variables as shown in Figure 7-7. Handled this way, it is immaterial how A and B come into the hopper.

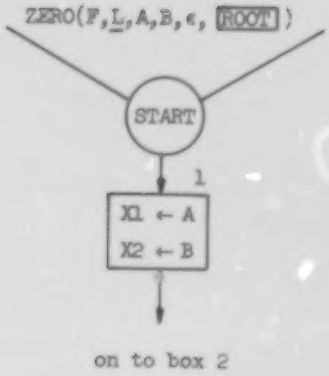


Figure 7-7. Another protection plan

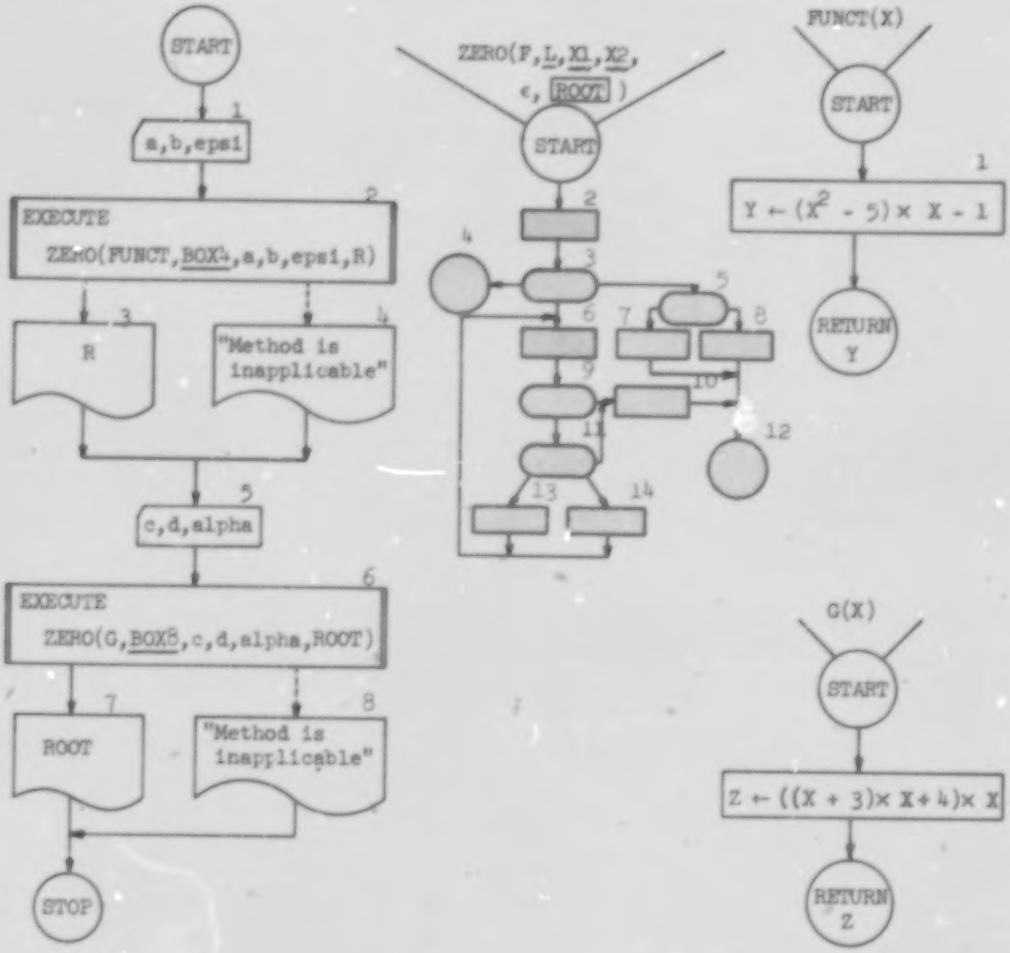
Now that we have completed our examination of the symbols in the funnel of ZERO we will briefly indicate the way in which this procedure is used.

First, we must be aware that there are three flow charts implicit in the preceding discussion. Besides the flow chart for the procedure ZERO there must be a main flow chart which calls for the execution of ZERO and at least one function reference flow chart for the function to which ZERO is to be applied. We illustrate in Figure 7-8 a simple case study. For this purpose we have created functions FUNCT and G defined by

$$FUNCT(x) = x^3 - 5x - 1$$

and

$$G(x) = x^3 + 3x^2 + 4x.$$


START
1
a,b,epsi
2
EXECUTE
ZERO(FUNCT,BOX4,a,b,epsi,R)
3
R
4
"Method is inapplicable"
5
c,d,alpha
6
EXECUTE
ZERO(G,BOX8,c,d,alpha,ROOT)
7
ROOT
8
"Method is inapplicable"
STOP
ZERO(F,L,X1,X2, ε, ROOT)
START
2
3
4
5
6
7
8
9
10
11
12
13
14
FUNCT(X)
START
1
Y ← (X² - 5) × X - 1
RETURN Y
G(X)
START
Z ← ((X + 3)× X+4)× X
RETURN Z

(a) Main (b) Procedure ZERO (c) Function references

Figure 7-8. Interplay

Very little explanation is needed for Figure 7-8. A main flow chart will usually do more with the roots than just print out the values. In particular, the main flow chart may call for the execution of zero several times, each time for the root of a different function.

F may be omitted from the funnel of Figure 7-8(b) (the silhouette of Figure 7-6) but then in Figure 7-6, the references to the function F are not to a place holder but to a function actually named F. In any main flow chart calling for the execution of ZERO, the function name must be omitted from the EXECUTE box and any main program can only call for the execution of ZERO applied to the single function put into the computer under the name of F.

Exercises 7-1 Set D

1. By graphing, it becomes clear that each of the two equations $\sin x = \frac{2}{3}x$ and $\tan x = 10x$ has a solution in the interval $[0, \frac{\pi}{2}]$. Prepare all the necessary flow charts for determining which of these roots is the greater. The ZERO procedure is understood to be available already in flow chart form.

2. By a certain theorem the function $H(X) = 6x^5 + 5x^4 - 4x^3 - 2x - 1$ has exactly one positive root. Since $H(0) = -1 < 0$ and $H(1) = 4 > 0$, this function has at least one root in the interval $(0,1)$. Draw the necessary flow charts which, when used together with ZERO, will print out the value, R, of this root and $G(R)$ where $G(X) = x^6 + x^5 - x^4 - x^2 - x - 1$ and 5 values of $G(X)$ and $H(X)$ separated by .01 in X on both sides of R.

3. Draw the flow charts which, used together with ZERO, will print out the root of $\ell n x = -x$. In this problem the flow chart for the reference function, ℓn, does not have to be drawn.

4. Suppose a satellite is in a circular orbit. Let the radius be 1 centered at the origin. Draw the necessary flow charts to be used in conjunction with ZERO for finding the intersections of the orbit with the trajectories

   (a) $xy = \frac{1}{4}$ for $0 \leq x \leq \frac{1}{2}$

   (b) $y = x^n$ where $n = 1, 2, 3, 4, 5.$
   for $0 \leq x \leq 1$

5. (For students with some knowledge of differential calculus) Find the minimum value of $x^5 - 3x^2 - 4x$ for $x \geq 0$. You may assume that the derivative has only one root.

6. While exploring the city, a visitor to San Francisco came upon a curious configuration of ladders in an alley between two tall buildings. Starting on opposite sides of the alley, each ladder was propped against the base of one building and crossed the alley to rest against a wall on the other side. As the visitor walked slowly beneath them, he mused to himself, "The rungs of the ladder are about one foot apart. One ladder looks about 25 feet long, while the second is a bit short of 20. The ladders cross about a foot over my head and I am six feet tall. I think I should be able to deduce how wide the alley is."

Your problem is to carry out this calculation for the visitor. To help you get started, study the two diagrams. The first is a schematic view of the alley. The second is a more abstract drawing of the situation. Take one ladder to be 25 feet long, the other 18.75, and take the crossover point to be 7.2 feet above the ground. Notice that there are two sets of similar right triangles involved, for which we can say

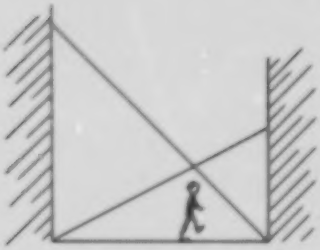


Diagram 1

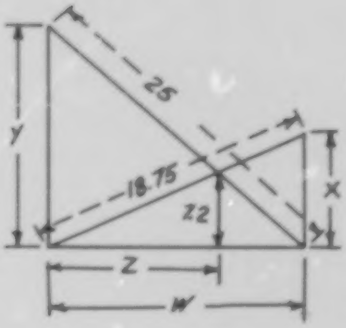


Diagram 2

$$\frac{7.2}{x} = \frac{z}{w} \qquad \text{and} \qquad \frac{7.2}{y} = \frac{w-z}{w} .$$

Substituting from the first equation, for $\frac{z}{w}$ in the second, we have

$$\frac{7.2}{y} = 1 - \frac{7.2}{x} .$$

Now using the Pythagorean Theorem we can rewrite this as

$$\frac{7.2}{\sqrt{25^2 - w^2}} = 1 - \frac{7.2}{\sqrt{(18.75)^2 - w^2}}$$

or

$$\frac{7.2}{\sqrt{25^2 - w^2}} + \frac{7.2}{\sqrt{18.75^2 - w^2}} - 1 = 0 .$$

Now we have the equation whose solution will give us the width of the alley. Why not call the function $f(w)$ and use the method of successive bisection? Take the starting interval as $[0,18]$ and find the width of the alley to the nearest foot following the flow chart of Figure 7-5.

7-2 The Area Under a Curve: An example, $y = 1/x$ between $x = 1$ and $x = 2$

In your previous work in mathematics you may have needed to find the area under a curve. Thus, if the curve $y = f(x)$ is plotted in the usual way, such as in Figure 7-9, we may want to calculate the area under $y = f(x)$, above the x-axis and between the vertical lines $x = a$ and $x = b$. We assume here that $f(x)$ is positive so that it makes sense to speak of area under the curve.

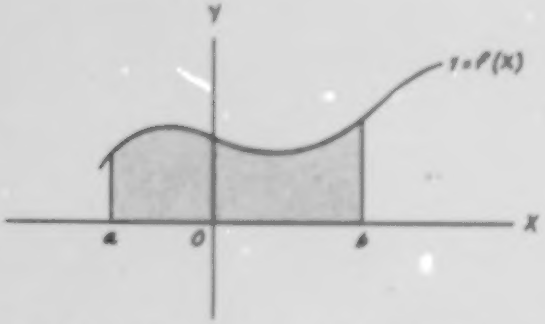


Figure 7-9. Graph showing the area under some curve $y = f(x)$ between $x = a$ and $x = b$

Area under $y = 1/x$ graphically

In many books the study of the logarithm function begins by introducing the natural logarithm of x, denoted by ℓnx as the area under the curve $y = 1/x$ between $x = 1$ and $x = x$. From this definition we are able to deduce the properties of ℓnx.

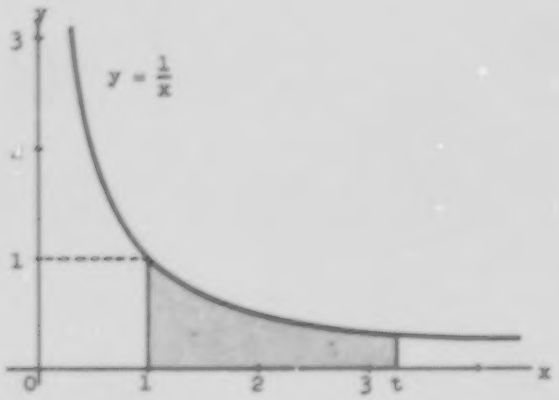


Figure 7-10. Graph of $y = \frac{1}{x}$ between $x = 1$ and $x = t$

Figure 2-10 shows the graph of $y = 1/x$. The shaded area under the curve between $x = 1$ and $x = t$ gives the value of ℓnx. If this graph is plotted on squared graph paper with a much larger scale, as in Figure 7-11, we can calculate the approximate value of ℓnt by counting squares in the area under the curve. For example, let us compute ℓn 1.05 approximately.

In Figure 7-11 we find that the number of squares lying wholely under the curve from $x = 1$ to $x = 1.8$ is 1438. Since the area of each square is $.02 \times .02 = .0004$ we see that the total area of these squares is .5732. This would clearly yield an underestimate for the area under the curve. If we had also counted the squares that the curve passes through, we would have obtained an overestimate for the area under the curve. The total number of squares with these additional squares counted in would be 1499 having a total area of $1499 \times .0004 = .5996$. The everage of these two estimates ought to yield a pretty good approximation of ℓn 1.8, the true area under the curve. This average is $(.5732 + .5996)/2 = .5864$. From tables the value of ℓn(1.8) correct to five decimal places is .58688.

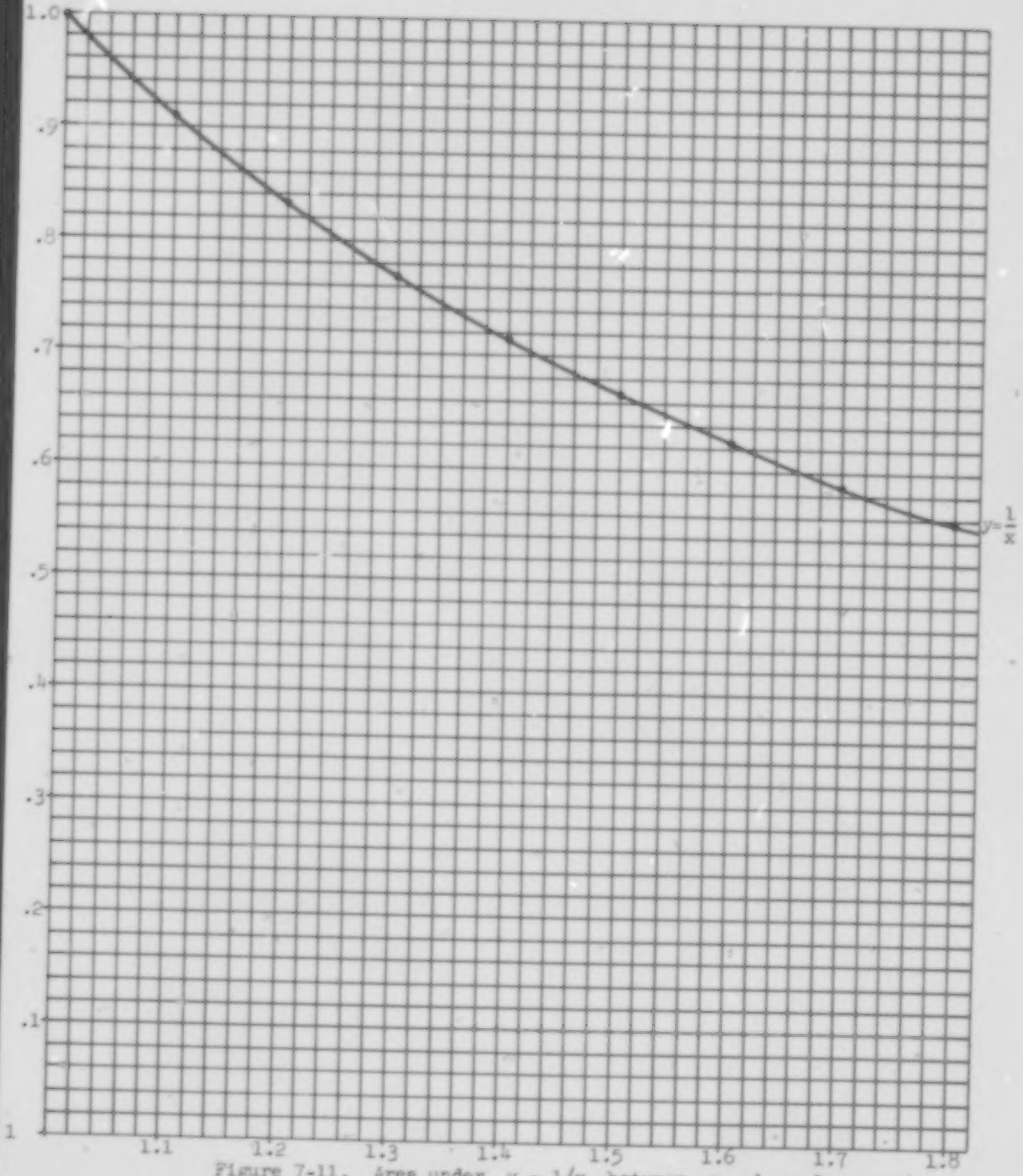


Figure 7-11. Area under $y = 1/x$ between $x = 1$ and $x = 1.8$ by adding coordinate squares

To be sure, there are other more accurate methods for calculating logarithms but we don't want to discuss these methods here. Using them, tables have been prepared and you are certainly acquainted with such tables.

Counting the squares under the curve is at best a tedious way of finding the area under the curve. We can find the area more easily and more accurately by making use of our knowledge of finding areas of certain figures. Thus, we know how to find the area of a rectangle or a triangle or a trapezoid. The area of a rectangle is, of course, the product of its length by its width; the area of a triangle is one-half the product of its base by its altitude; and the area of a trapezoid is the product of the altitude by the average of its two parallel bases, as shown below.

$$\text{Area} = h \times \frac{b_1 + b_2}{2}$$

Area under $y = 1/x$ by summing trapezoidal areas

Now suppose we want to calculate an approximation to $\ln 2$. We know that $\ln 2$ is the area under the curve $y = 1/x$ between $x = 1$ and $x = 2$. For convenience in writing we denote the equation of this curve by $y = f(x)$. Here, of course, $f(x) = 1/x$. This is the area ABDC in Figure 7-7.

We can calculate an approximation to this area if we draw a straight line from C to D as shown in Figure 7-12. Then we have a trapezoid whose area

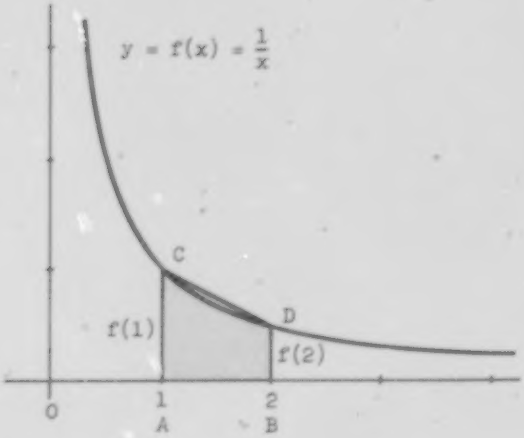


Figure 7-12. Area under $y = 1/x$ approximated by a single trapezoid

is only slightly larger than the desired area. The area of the trapezoid is

$$T_0 = h \times \left(\frac{b_1 + b_2}{2}\right)$$

$$= (B - A) \times \frac{f(A) + f(B)}{2}$$

$$= (2 - 1) \times \frac{f(1) + f(2)}{2}$$

$$= \frac{1}{2} \times (f(1) + f(2))$$

$$= \frac{1}{2} \times \left(1 + \frac{1}{2}\right) = \frac{1}{2} \cdot \frac{3}{2} = \frac{3}{4} .$$

So we know that the desired area, $\ell n2$, is less than .75 but fairly close to it.

We can improve this estimate of the area if we divide it into two parts by drawing the vertical line $x = \frac{3}{2}$ in Figure 7-13.

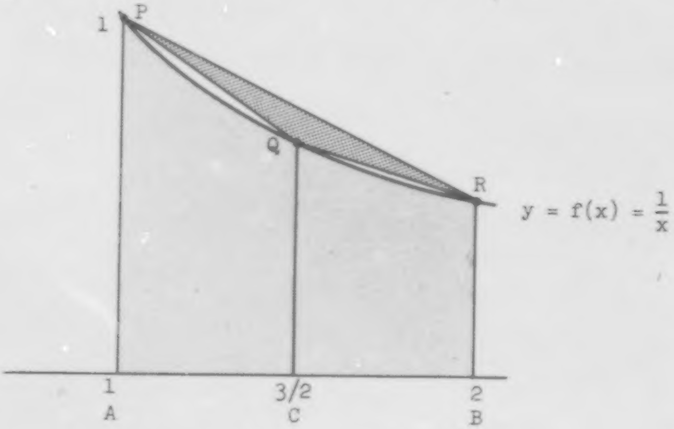


Figure 7-13. Approximation of $\ell n2$ using two trapezoids

This figure allows us to compare geometrically the approximation using one trapezoid with the approximation using two trapezoids. The amount by which the error is reduced is represented by the area of the shaded triangle PQR. All that is left of the error is represented by the two tiny unshaded slivers between the two shaded regions.

Exercises 7-2 Set A

1. The area of the trapezoid ABRP has already been calculated to be $\frac{3}{4}$ and is called T_0. Calculate the value of T_1, the sum of the areas of the two trapezoids ACQP and CBRQ.

2. Find the value of the absolute difference $|T_0 - T_1|$ (the area of the shaded triangle PQR).

Our calculations of the sum of the areas of the two trapezoids now follows.

$$\text{Area of trapezoid ACQP} = \frac{1}{2} \times \frac{f(1) + f(\frac{3}{2})}{2} = \frac{1}{2}[\frac{1}{2}f(1) + \frac{1}{2}f(\frac{3}{2})].$$

$$\text{Area of trapezoid CBRQ} = \frac{1}{2} \times \frac{f(\frac{3}{2}) + f(2)}{2} = \frac{1}{2}[\frac{1}{2}f(\frac{3}{2}) + \frac{1}{2}f(2)].$$

The sum of these values is

$$T_1 = \frac{1}{2}[\frac{1}{2}\, f(1) + f(\frac{3}{2}) + \frac{1}{2}\, f(2)].$$

$$= \frac{1}{2}[\frac{1}{2}(f(1) + f(2)) + f(\frac{3}{2})].$$

And since T_0 has already been seen to be $\frac{1}{2}(f(1) + f(2))$ we can write T_1 in the form

$$T_1 = \frac{1}{2}[T_0 + f(\frac{3}{2})] = \frac{1}{2}[\frac{3}{4} + \frac{2}{3}] = \frac{17}{24} \cong .708 .$$

Note well that T_1 is calculated from T_0 using only one new value of the function; namely, $f(\frac{3}{2})$.

We could get a still better approximation to the area between 1 and 2 if we divided this area into 3 parts by equally spaced vertical lines or better still, by dividing it into 4 parts by equally spaced vertical lines. The latter is particularly convenient as we will then merely add two more points of subdivision, $\frac{5}{4}$ and $\frac{7}{4}$, for the interval AB to the one, $\frac{3}{2}$, we already have. The advantage of using subdivision points previously used should be obvious for by this method we can use the functional values previously calculated.

We show in Figure 7-14 the result of halving each of the intervals AC and CB of Figure 7-13, thereby approximating $\ell n2$ by four trapezoids.

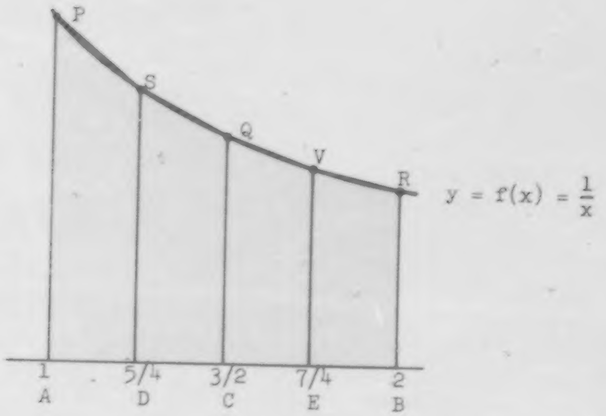


Figure 7-14. Divide and conquer!

This time the areas of the four slivers representing the error is virtually imperceptible to the unaided eye.

Exercises 7-2 Set B

1. Calculate the sum T_2 of the areas of the four trapezoids ADSP, DCQS, CEVQ and EBRU of Figure 7-11.
2. Calculate the absolute difference of T_1 and T_2 in exact fractional form and also give a four digit decimal approximation of this difference.
3. Make a careful tracing of Figure 7-14 shading the two triangles PQS and QRV. Comparing with Figure 7-13, show that the sum of the areas of these two triangles is equal to the value of $|T_1 - T_2|$ calculated in Problem 2.

We will present the calculation of T_2, the sum of the areas of the four trapezoids, in a revealing form:

$$\text{Area ADSP} = \frac{1}{4}\left(\frac{f(1) + f(\frac{5}{4})}{2}\right) \quad \text{Area DCQS} = \frac{1}{4}\left(\frac{f(\frac{5}{4}) + f(\frac{3}{2})}{2}\right) .$$

$$\text{Area CEVQ} = \frac{1}{4}\left(\frac{f(\frac{3}{2}) + f(\frac{7}{4})}{2}\right) \quad \text{Area EBRV} = \frac{1}{4}\left(\frac{f(\frac{7}{4}) + f(2)}{2}\right) .$$

Their sum is

$$T_2 = \frac{1}{4}[\frac{1}{2} f(1) + f(\tfrac{5}{4}) + f(\tfrac{3}{2}) + f(\tfrac{7}{4}) + \frac{1}{2} f(2)] .$$

On rearranging these terms we have

$$T_2 = \frac{1}{4}[(\frac{1}{2} f(1) + f(\tfrac{3}{2}) + \frac{1}{2} f(2)) + f(\tfrac{5}{4}) + f(\tfrac{7}{4})]$$

$$= \frac{1}{2}[\frac{1}{2}(\frac{1}{2} f(1) + f(\tfrac{3}{2}) + \frac{1}{2} f(2)) + \frac{f(\frac{5}{4}) + f(\frac{7}{4})}{2}] .$$

Inside the brackets we see the expression $\frac{1}{2}(\frac{1}{2} f(1) + f(\frac{3}{2}) + \frac{1}{2} f(2))$ which is exactly one of the forms calculated for T_1. Therefore,

$$T_2 = \frac{1}{2}[T_1 + \frac{f(\frac{5}{4}) + f(\frac{7}{4})}{2}]$$

$$= \frac{1}{2}[\frac{17}{24} + \frac{\frac{4}{5} + \frac{4}{7}}{2}] = \frac{1171}{1680} \cong .697 .$$

The tabulated value of $\ell n 2$ to four decimal places is .6931.

From the previous calculations a pattern emerges for computing the successive values of the approximations $T_0, T_1, T_2, \ldots$. In each case we add to the previous approximation the arithmetic mean (or average) of the newly computed fractional values and then halve the result. Thus,

$$T_3 = \frac{1}{2}[T_2 + \frac{f(\frac{9}{8}) + f(\frac{11}{8}) + f(\frac{13}{8}) + f(\frac{15}{8})}{4}]$$

and so on. It is easily checked that this agrees with the value of T_3 calculated directly (i.e., without expressing it in terms of T_2) which is given by

$$T_3 = \frac{1}{8}[\frac{1}{2}f(1) + f(\tfrac{9}{8}) + f(\tfrac{5}{4}) + f(\tfrac{11}{8}) + f(\tfrac{3}{2}) + f(\tfrac{13}{8}) + f(\tfrac{7}{4}) + f(\tfrac{15}{8}) + \frac{1}{2}f(2)].$$

In the general case we assume that the interval from 1 to 2 is divided into 2^n equal parts by the points

$$1,\ 1+\frac{1}{2^n},\ 1+\frac{2}{2^n},\ \ldots,\ 1+\frac{k}{2^n},\ \ldots,\ 1+\frac{2^n-1}{2^n},\ 2\ .$$

Every other point in this list is a newly adjoined partition point. That is, the new partition points are

$$1+\frac{1}{2^n},\ 1+\frac{3}{2^n},\ 1+\frac{5}{2^n},\ \ldots,\ 1+\frac{2^n-1}{2^n}\ .$$

And now, according to the above pattern,

$$T_n = \frac{1}{2}\left[T_{n-1} + \frac{f(1+\frac{1}{2^n}) + f(1+\frac{3}{2^n}) + \ldots + f(1+\frac{2^n-1}{2^n})}{2^{n-1}}\right].$$

We now present a preliminary flow chart showing how one could calculate the approximations T_0, T_1, T_2, ..., T_n ... for $1, 2, 4, \ldots, 2^n, \ldots$ subdivisions. We do not provide any way of terminating the calculations yet.

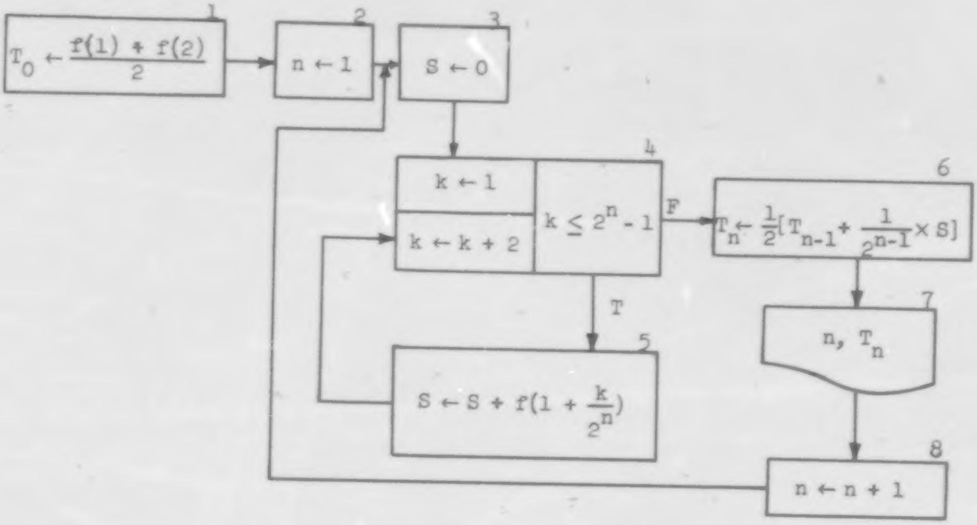

1
$T_0 \leftarrow \frac{f(1)+f(2)}{2}$
2
$n \leftarrow 1$
3
$S \leftarrow 0$
4
$k \leftarrow 1$
$k \leftarrow k+2$
$k \le 2^n - 1$
F
T
5
$S \leftarrow S + f(1+\frac{k}{2^n})$
6
$T_n \leftarrow \frac{1}{2}[T_{n-1} + \frac{1}{2^{n-1}} \times S]$
7
n, T_n
8
$n \leftarrow n+1$

Figure 7-15. Non-terminating area calculation for $y = 1/x$

The calculation described in Figure 7-15 will never terminate. We must now introduce a measure of accuracy. We agree to stop the calculation as soon as the magnitude of the difference between 2 consecutive sums is small, say, less than some given ϵ. If we incorporate this test into the flow chart of Figure 7-15, we obtain for the complete calculation of the area under a curve $y = f(x)$ in the interval between $x = 1$ and $x = 2$ the flow chart given in

Figure 7-16. Of course, it is understood that in the example under discussion $f(x)$ is given by $f(x) = \frac{1}{x}$.

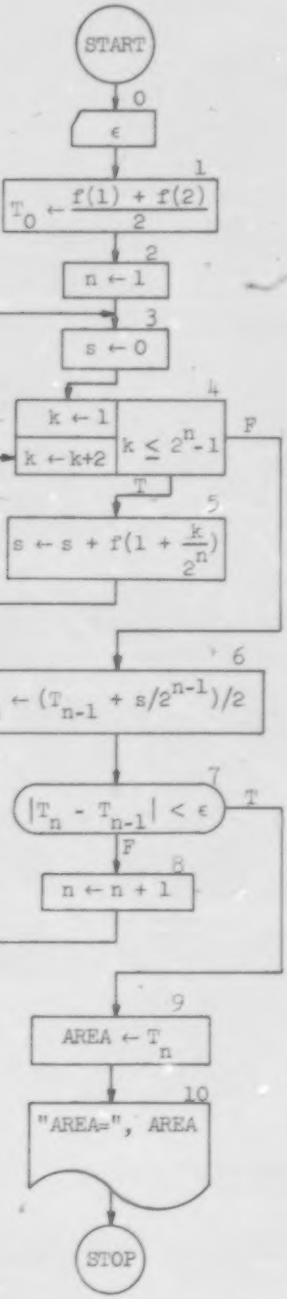

START
0
ϵ
1
$T_0 \leftarrow \frac{f(1) + f(2)}{2}$
2
$n \leftarrow 1$
3
$s \leftarrow 0$
4
$k \leftarrow 1$
$k \leftarrow k+2$
$k \leq 2^n - 1$
F
T
5
$s \leftarrow s + f(1 + \frac{k}{2^n})$
6
$T_n \leftarrow (T_{n-1} + s/2^{n-1})/2$
7
$|T_n - T_{n-1}| < \epsilon$
T
F
8
$n \leftarrow n + 1$
9
$AREA \leftarrow T_n$
10
"AREA=", AREA
STOP

Figure 7-16. Complete flow chart for area under $y = \frac{1}{x}$ between $x = 1$ and $x = 2$ (first version)

Exercises 7-2 Set C

1. Find the abscissa values resulting from the division of the segment (1,0) to (2,0) on the x-axis into 16 equal parts.

2. If an interval is divided into 2^n equal parts, what happens to the number of subdivisions when n is increased by 1 ?

3. Find the abscissa values resulting from the division of the segment (1,0) to (5,0) on the x-axis into eight equal parts. (Note: Let $b - a = h$ and the points $1, 1 + h/2^n, 1 + 2h/2^n \ldots$).

4. Given the function $y = 3x^2 + 2x + 1$ and the interval $x = 1$ to $x = 2$, you are to approximate the area below the curve and above the x-axis. Decide what changes are necessary in Figure 7-16 in order to specialize the approximating process to this function and this interval. Draw the flow chart for this function in the interval.

5. Study the flow chart of Figure 7-16 for the function $y = x^2$ in the interval $x = 1$ to $x = 2$.

   (a) What is the value of T_0 ?

   (b) When $n = 1$ how many times do you enter the iteration box, 4 ? How many times do you exit to box 5? How many times to box 6?

   (c) What is the value of T_n in box 6 the first time you calculate it?

   (d) When $n = 3$ how many times do you enter the iteration box 4 ?

   (e) When $n = 3$, what is the value of S the last time you enter the iteration box 4 ?

   (f) When $n = 10$, how many times do you enter box 4 ? How many of these times do you exit to box 5 ?

   (g) For $\epsilon = .03$ calculate the value of n, when box 9 is entered.

6. Suppose that the interval from 1 to 2 is divided into n equal parts by points $1, 1 + \frac{1}{n}, 1 + \frac{2}{n}, \ldots, 1 + \frac{k}{n}, \ldots, 1 + \frac{n-1}{n}, 2$ where it is not assumed that n is a power of 2. The area under the curve $y = f(x)$ between $x = 1$ and $x = 2$ is to be approximated by a sum of areas of trapezoids.

   (a) Obtain a formula for this approximation in terms of the ordinates at the points of division. (Look back in the previous discussion and note how T_0, T_1, T_2 and T_3 were expressed in terms of the ordinates at the points of subdivision.)

   (b) Draw a flow chart to describe the calculation.

7. (a) Sketch the curve $y = \frac{1}{x}$ between $x = 1$ and $x = 9$.

   (b) T_0: Approximate the area by a single trapezoid bounded by $f(1)$ and $f(9)$. First express the area in terms of $f(1)$ and $f(9)$; then evaluate.

   (c) T_1: Approximate the area by the sum of 2 trapezoids, one bounded by $f(1)$ and $f(5)$, the other by $f(5)$ and $f(9)$. Express in terms of $f(1)$, $f(5)$ and $f(9)$. Evaluate.

   (d) T_2: Approximate by 4 trapezoids. State in terms of $f(1)$, $f(3)$, $f(5)$, $f(7)$ and $f(9)$. Evaluate.

   (e) T_3: Approximate by 8 trapezoids. Again express in $f(1)$, etc., and evaluate.

   (f) Look at the formula for T_1. How can it be derived from T_0 and $f(5)$. How can T_2 be found from T_1; T_3 from T_2? Use these new formulas to verify your calculations in (c), (d), and (e).

   (g) From your study of part (f) write a formula for T_4. Can you derive one for T_n ?

   (h) Sketch a flow chart to evaluate the area. It should terminate as soon as 2 successive approximations differ in absolute value by less than 10^{-3}.

8. Repeating the method of problem 7, calculate the area below $y = x^2$, above x-axis, between $x = 2$ and $x = 4$.

   (a) Evaluate T_0, T_1, T_2.

   (b) Relate T_1 to T_0 and T_2 to T_1.

   (c) Write a formula for T_3, for T_n.

Another look at the flow chart for the area under $y = 1/x$

We note that there are several sources of inefficiency in this calculation. In our iteration box we have to test k against $2^n - 1$ which will require a calculation of $2^n - 1$ several times. Even one calculation of 2^n for each n may be time-consuming and should be avoided if possible. It will be better to use the test $k < 2^n$ instead of $k \leq 2^n - 1$. Then we can simplify this to $k < m$ where we set $m = 1$ when $n = 0$ at the start and double m each time n is increased by 1. Thus, m represents the quantity 2^n.

In the same way we can avoid the divisions by 2^n and 2^{n-1}. Instead, we keep a number h which is set equal to 1 for $n = 0$ and then divided by 2 each time as n is increased by 1. Thus, at each stage $h = 1/m = 1/2^n$.

We do not really need to keep all the sums $T_0, T_1, T_2, \ldots, T_n$ but actually need only the current sum and the previous sum at each stage. We could denote these respectively, NUAREA and OLDAREA. Moreover, we have no further need for the variable n. If we do these things, then instead of Figure 7-16 we would have the more efficient flow chart, Figure 7-17.

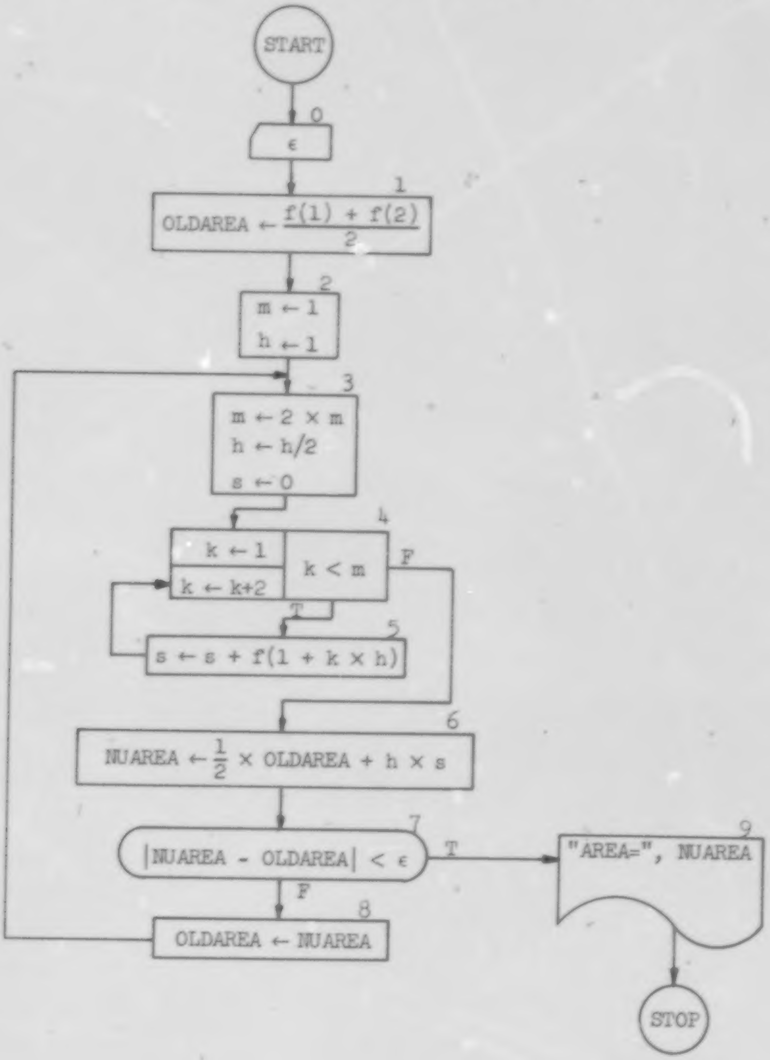

START
0
ε
1
OLDAREA ← (f(1) + f(2))/2
2
m ← 1
h ← 1
3
m ← 2 × m
h ← h/2
s ← 0
4
k ← 1
k ← k+2
k < m
F
T
5
s ← s + f(1 + k × h)
6
NUAREA ← 1/2 × OLDAREA + h × s
7
|NUAREA - OLDAREA| < ε
T
F
8
OLDAREA ← NUAREA
9
"AREA=", NUAREA
STOP

Figure 7-17. Improvement of Figure 7-16 flow chart

7-3 The Area Under a Curve: The General Case

In the previous section and in its exercises we have seen how to calculate approximations to the area under various curves. We now want to discuss the formulas and the necessary flow charts for solving the same problem for any curve which lies above the x-axis.

We want to find an approximation to the area under a curve $y = f(x)$, above the x-axis and between the vertical lines $x = a$ and $x = b$. See Figure 7-18.

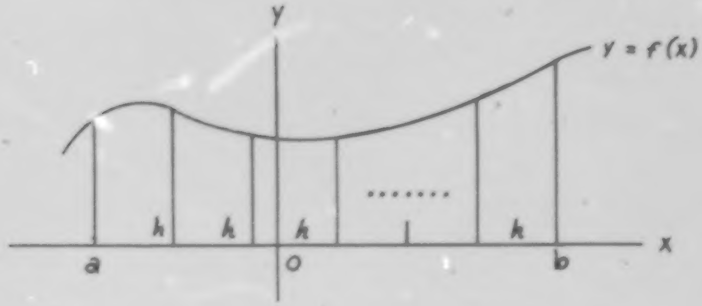


Figure 7-18. Curve $y = f(x)$ between $x = a$ and $x = b$ showing n subintervals

We can divide the interval from a to b into n equal parts and approximate the area standing on each of these parts by a trapezoid as shown in Figure 7-19.

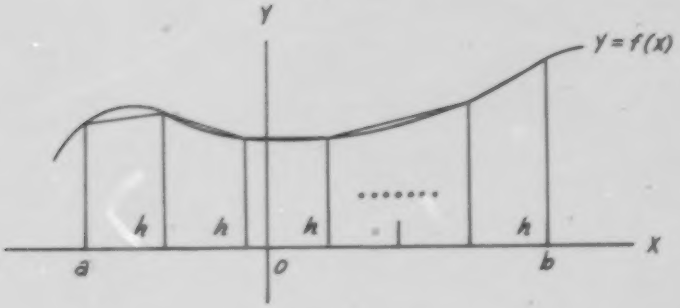


Figure 7-19. n subintervals with trapezoids

We denote the length of each subinterval by h. Clearly,

$$h = (b-a)/n.$$

Then the x-coordinates of the points of division are

$$a,\ a + h,\ a + 2h,\ \ldots,\ a + kh,\ \ldots,\ a + (n-1)h,\ b.$$

The ordinates at these points are, respectively,

$$f(a),\ f(a+h),\ f(a+2h),\ \ldots,\ f(a+kh),\ \ldots,\ f(a + (n-1)h),\ f(b).$$

Then the areas of the approximating trapezoids are

$$h \cdot \frac{f(a) + f(a+h)}{2},\ h \cdot \frac{f(a+h) + f(a+2h)}{2},\ \ldots,$$
$$h \cdot \frac{f(a+kh) + f(a+(k+1)h)}{2},\ \ldots,\ h \cdot \frac{f(a+(n-1)h) + f(b)}{2}.$$

The sum, S_n, of these areas is given by

$$S_n = h[\tfrac{1}{2}f(a) + f(a+h) + f(a+2h) + \ldots + f(a+kh) + \ldots + f(a+(n-1)h) + \tfrac{1}{2}f(b)]$$

where, of course, $h = (b - a)/n$.

In order to decide what value of n to use to obtain a specified accuracy in the calculation it will be convenient as in Section 7-2 to calculate the approximating sum for $1, 2, 4, 8, 16, \ldots$ subdivisions of the interval from a to b. Then, as in the example described in that section, each sum can be calculated from the previous one and the function $f(x)$ has to be evaluated only at the new points of subdivision. Let T_0 denote the sum when the interval a to b is taken without subdividing, T_1 the sum with 2 subintervals, T_2 the sum with 4 subintervals, ..., T_n the sum with 2^n subintervals. Then we have

$$T_0 = h_0\frac{f(a) + f(b)}{2}$$

$$T_1 = h_1[\tfrac{1}{2}f(a) + f(a+h_1) + \tfrac{1}{2}f(b)]$$

where $h_0 = b - a$, $h_1 = (b-a)/2 = h_0/2$. If we compare the formulas for T_0 and T_1 and use the formula for h_1, we see that

$$T_1 = \tfrac{1}{2}[T_0 + h_0f(a+h_1)] = \tfrac{1}{2}T_0 + h_1f(a+h_1).$$

We have also

$$T_2 = h_2[\tfrac{1}{2}f(a) + f(a+h_2) + f(a+2h_2) + f(a+3h_2) + \tfrac{1}{2}f(b)]$$

where $h_2 = (b-a)/4 = h_1/2$. If we compare the formulas for T_1 and T_2 we see, since $a + 2h_2 = a + h_1$, that

$$T_2 = \tfrac{1}{2}(T_1 + h_1[f(a+h_2) + f(a+3h_2)]) = \tfrac{1}{2}T_1 + h_2[f(a+h_2) + f(a+3h_2)].$$

The generating principle is now clear. We can write immediately

$$T_3 = \tfrac{1}{2} T_2 + h_3[f(a+h_3) + f(a+3h_3) + f(a+5h_3) + f(a+7h_3)]$$

where $h_3 = (b-a)/8 = h_2/2$. In the general case we have

$$T_n = \tfrac{1}{2} T_{n-1} + h_n[f(a+h_n) + f(a+3h_n) + \ldots + f(a + (2^n-1)h_n)]$$

$$= \tfrac{1}{2} T_{n-1} + h_n \sum_{k=1}^{2^{n-1}} f(a + (2k-1)h_n)$$

where $h_n = (b-a)/2^n = h_{n-1}/2$.

Finally we must provide a means of terminating the calculation of these approximating sums. We agree to stop the calculation as soon as the absolute value of the difference between two consecutive calculated sums is small, say, less than some given ϵ. We try to make the calculation reasonably efficient by avoiding calculation of 2^n and we are thus led to a flow chart similar to that of Figure 7-17. This flow chart, Figure 7-20, describes the calculation of an approximation to the area under the curve $y = f(x)$, above the x-axis and between $x = a$ and $x = b$. We assume that the necessary values of $f(x)$ will be supplied by a functional reference flow chart. As in the flow chart of Figure 7-17, the use of a subscript n is avoided since we need only the most recent values of h and m and the last two values of T (called OLDAREA and NUAREA).

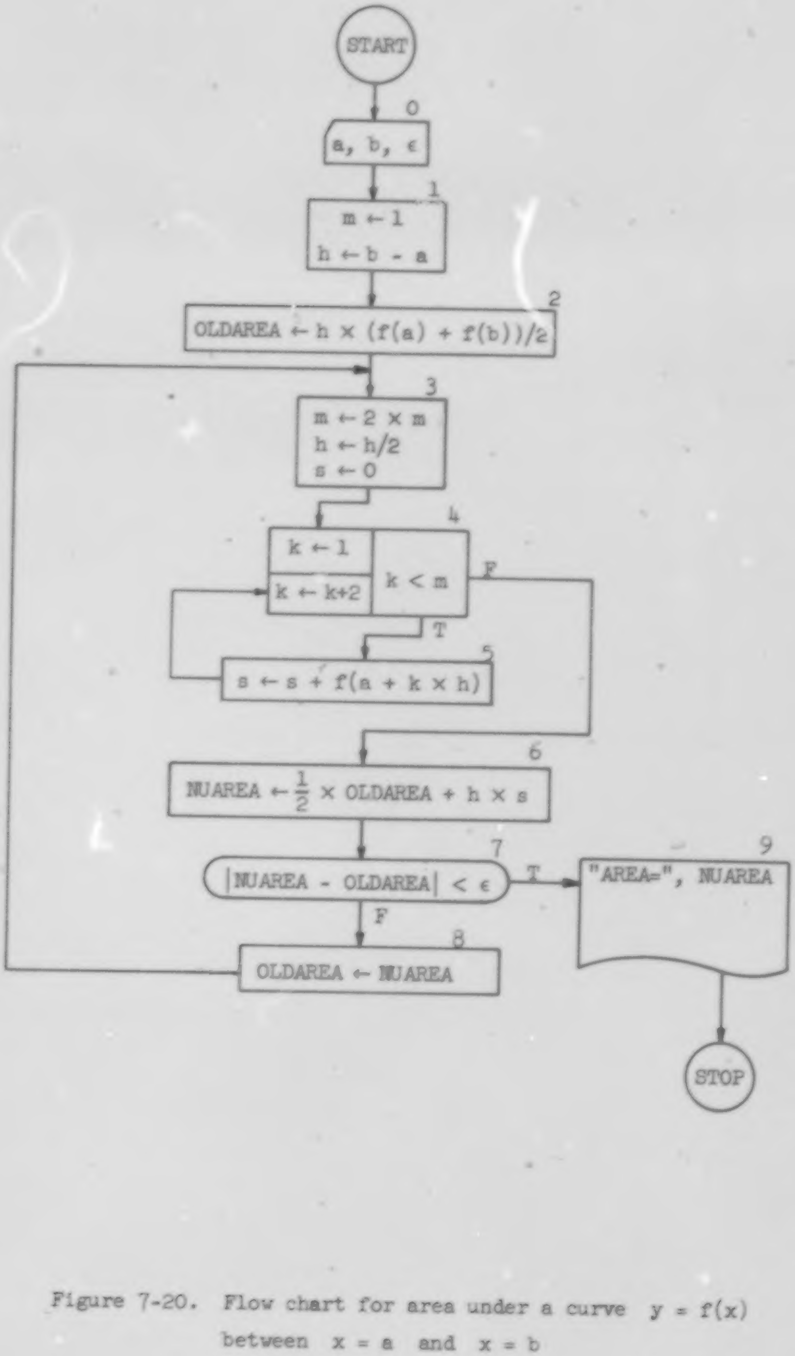
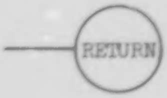

START
0
a, b, ε
1
m ← 1
h ← b - a
2
OLDAREA ← h × (f(a) + f(b))/2
3
m ← 2 × m
h ← h/2
s ← 0
4
k ← 1
k ← k+2
k < m
F
T
5
s ← s + f(a + k × h)
6
NUAREA ← 1/2 × OLDAREA + h × s
7
|NUAREA - OLDAREA| < ε
T
F
8
OLDAREA ← NUAREA
9
"AREA=", NUAREA
STOP

Figure 7-20. Flow chart for area under a curve $y = f(x)$ between $x = a$ and $x = b$

It will be interesting to compare the very minor points of difference in the flow charts of Figures 7-17 and 7-20. These differences are seen in boxes 0, 1, 2, 5. If a and b of Figure 7-20 are replaced by 1 and 2, respectively, then Figure 7-20 is identical with Figure 7-17.

Naturally, we would like to be able to file away the flow chart of Figure 7-20 to use it as a procedure in finding the area under any curve over any interval [a,b]. To do this we need make changes only in the flow chart of Figure 7-20. We replace box 9 and its stop box by



.

And we replace the start box and box 0 by either 7-21(a) or 7-21(b) according to whether function names are available to us as parameters.

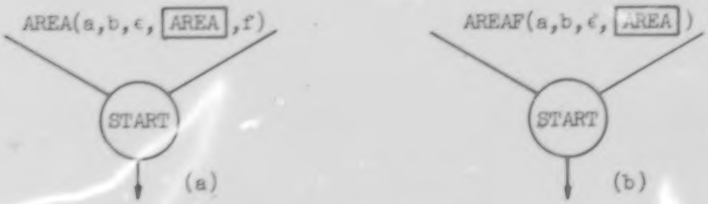


Figure 7-21. Two procedurizations of Figure 7-[illegible]

In Figure 7-21(b) the procedure will apply only to a function named f. The framing of the variable AREA in the funnels follows the convention introduced in Section 7-1.

Exercises 7-3

1. Draw a flow chart to represent the calculation

$$S_n = h\left(\frac{1}{2}f(a) + \sum_{k=1}^{n-1} f(a+kh) + \frac{1}{2}f(b)\right)$$

where $h = (b-a)/n$.

2. Draw a flow chart similar to Figure 7-15 to indicate the calculation of the $T_0, T_1, T_2, \ldots, T_n, \ldots$ where

$$T_n = \frac{1}{2}T_{n-1} + h_n \sum_{k=1}^{2^{n-1}} f(a + (2k-1)h_n)$$

and $h_n = (b-a)/2^n$. Do not provide any termination.

3. It might happen that the calculation described in Figure 7-20 would never terminate because the function $f(x)$ might be badly behaved or because the tolerance ϵ is chosen too small for the accuracy of the machine used for the calculation. In such a case we say that the calculation has entered an endless loop. Suggest a way to protect against this possibility and revise the flow chart of Figure 7-20 to incorporate this protection. Draw it as a procedure and include the necessary statement label in the funnel.

4. Compare the flow charts drawn for Exercises 7-2, Set C, Problem 8(h), with the flow chart of Figure 7-20. If they are different, revise them to agree with the flow chart of Figure 7-20.

5. For the areas described below, calculate approximations following the steps of the flow chart of Figure 7-20.

   (a) Below $y = x^2$, above the x-axis, between $x = 2$ and $x = 4$. Use $\epsilon = 0.1$. (True area is $56/3$.)

   (b) Below $y = x^3$, above $y = x^2$, between $x = 1$ and $x = 4$. Use $\epsilon = 0.5$. (True area is $171/4$.)

6. Draw a functional reference flow chart which can be used with the procedure of Figure 7-20 (with one of the funnels of Figure 7-21) to approximate the value of π to four decimal places. Draw, also, the flow chart which calls on the procedure and prints out the result.

7. The number x for which $\ln x = 1$ is a very important mathematical constant on a par with π. This constant is designated by the letter e. It is interesting that we now have a method at our disposal (although, not the best one) for computing the value of e. This method is based on the fact that e is the root of

$$\ln x - 1 = 0.$$

Thus, if we can prepare a flow chart which will compute the values of $\ln x - 1$ to, say, six decimal places, we can then apply the procedure ZERO to find this root.

Make the necessary revisions in the flow chart of Figure 7-20 to convert it into a functional reference flow chart for $F(X) = \ln x - 1$. You will have to decide what to do about a, b, ϵ and f occurring in Figure 7-20. The main flow chart which calls on ZERO will involve the use of some preliminary estimates of the interval in which the root lies.

7-4 Simultaneous Linear Equations: Developing a systematic method of solution

Introduction: Solution by graphing

You have often solved systems of two equations. You may have also solved systems of three or even four simultaneous linear equations. In many problems of science, engineering, business, politics, etc., it is necessary to solve systems of simultaneous linear equations involving very large numbers of variables and equations, perhaps as many as 10,000 equations involving 10.000 variables. It is therefore important to study the problem of solving systems of linear equations and to devise efficient methods for solving such systems.

You will recall that the problem of solving two simultaneous linear equations in x and y can be interpreted geometrically. Thus, an equation such as

$$3x - 4y = 12$$

represents a straight line which can be plotted on a set of xy-axes. Another equation such as

$$4x + 12y + 23 = 0$$

also represents a straight line. If we are asked to solve these equations simultaneously, then we are seeking values of x and y which satisfy both equations at the same time. Geometrically this means that the point whose coordinates are these values of x and y lies on both lines and hence is their intersection. The graphs of these two lines are shown in Figure 7-22.

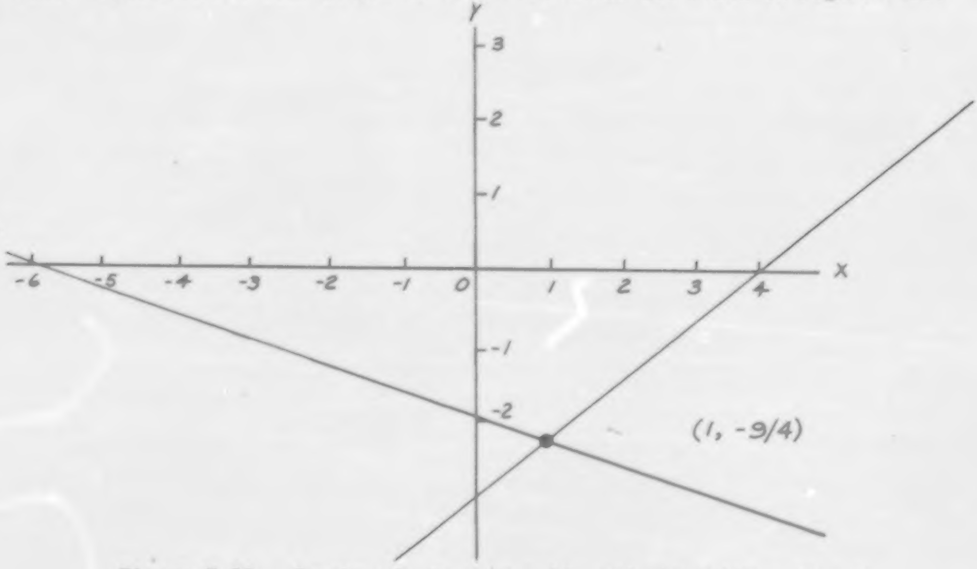


Figure 7-22. Graphs of $3x - 4y = 12$ and $4x + 12y + 23 = 0$

We see that the lines intersect in the point $(1, -9/4)$ and thus $x = 1$, $y = -9/4$ is the solution of this pair of equations.

We know that two lines always intersect in exactly one point unless they are parallel or are really the same line. Thus, two simultaneous linear equations also have exactly one solution unless they represent parallel lines or the same line. If the lines are parallel, there is no solution. For example, the equations

$$x + 2y = 7$$
$$x + 2y = 8$$

represent parallel lines and hence have no solution. This situation is shown in Figure 7-23.

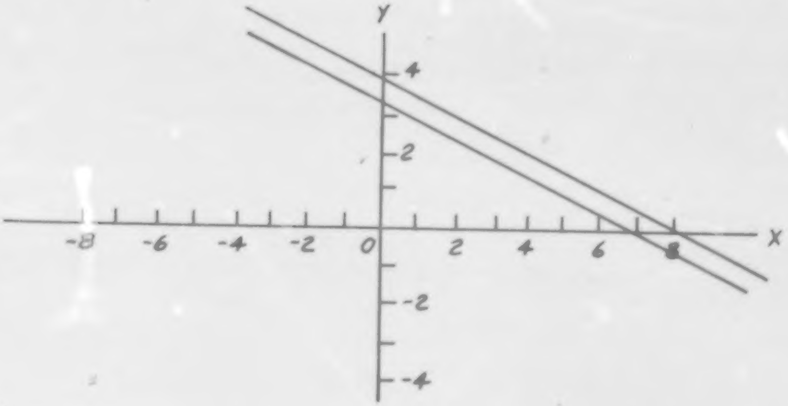


Figure 7-23. Graphs of $x + 2y = 7$ and $x + 2y = 8$

On the other hand, the equations

$$x + 2y = 7$$
$$2x + 4y = 14$$

have infinitely many solutions as they are really the same line, namely, the lower line of Figure 7-23.

You are also familiar with the fact that a linear equation in three variables can be represented as a plane in 3-dimensional space. Hence, three simultaneous linear equations in three unknowns will have a unique solution if the corresponding planes intersect in a single point and this solution will be their point of intersection. But it is also possible that three planes intersect in one, two or three straight lines, or that they be parallel or even that they all be really the same plane. Thus, three simultaneous linear equations in three unknowns may also possess infinitely many solutions or no solutions. Similar situations arise when we consider more equations in more unknowns.

Exercises 7-4 Set A

Draw the graphs of each pair of straight lines on a separate set of axes and determine their intersection, if any.

1. $4x - 2y = 7$
   $3x + 5y = 13$

2. $3x - 5y = 15$
   $5x + 3y = 12$

3. $3x - y = 7$
   $6x - 2y = 11$

4. $x + 3y = 11$
   $3x + 9y = 33$

A Systematic Method of Solution

In this section we are going to assume that the system of equations has one solution and that no special difficulties arise. You have learned now to solve two equations in two unknowns by several different methods. In one common method, often called "substitution," you solve for one variable, say y, in one equation and substitute it into the other equation, thus eliminating y. For example, given the set

$$3x - 4y = 12$$

$$4x + 12y = -23$$

we might solve the first equation for y, obtaining

$$y = \frac{1}{4}(3x-12)$$

and substitute this into the second equation, obtaining

$$13x = 13.$$

Hence, $x = 1$ and it follows that $y = -9/4$.

Another method, sometimes called "addition and subtraction" consists of adding (or subtracting) a multiple of one equation to (or from) the other equation so as to eliminate one of the variables. Using the same set of equations as above we might add 3 times the first equation to the second equation yielding the equivalent set of equations,

$$3x - 4y = 12$$

$$13x = 13 .$$

Again we find that $x = 1$ and from the first equation $y = -9/4$.

It is important to point out that when we modify an equation in a system of simultaneous linear equations by adding to (or subtracting from) it a multiple of another equation of the system, we do not change the solution set of the system.

In Chapter 6 we described an algorithm for the solution of two equations in two unknowns which was more suitable than either of these methods for use on an automatic digital computer because it was more systematic. It did not depend on an examination of the equations to pick out "convenient" coefficients. For our hand calculations we attempted to choose the order of calculations to minimize the work. But since the computer calculates with big numbers or fractional numbers just as easily as with small integers, this searching and choice is no longer necessary. If we follow the method described in Chapter 6, then we always proceed in the same systematic manner. If we apply this method to our system of equations

$$3x - 4y = 12$$

$$4x + 12y = -23$$

then we begin by dividing the first equation by 3 obtaining

$$x - \frac{4}{3}y = 4.$$

We then subtract 4 times this equation from the second equation in order to eliminate x from the latter. Thus, we obtain

$$(12 + \frac{16}{3})y = -23 - 16$$

or

$$\frac{52}{3}y = -39.$$

This equation is now divided by $52/3$ reducing it to the form

$$y = -9/4.$$

From this equation it is obvious that the solution for y is $-9/4$ as before. We now substitute this value of y into the modified first equation obtaining $x = 1$.

In order to make clear how this systematic method can be applied to larger systems of equations we shall use it to solve a system of three simultaneous linear equations. Consider the system

$$\begin{aligned} 3x + 2y + 7z &= 4 \\ 2x + 3y + z &= 5 \\ 3x + 4y + z &= 7. \end{aligned} \tag{1}$$

We begin by dividing the first equation by 3, obtaining the system

$$\begin{aligned} x + \frac{2}{3}y + \frac{7}{3}z &= \frac{4}{3} \\ 2x + 3y + z &= 5 \\ 3x + 4y + z &= 7. \end{aligned} \tag{2}$$

We must now eliminate x from the succeeding equations. Hence, we subtract twice the modified first equation (2) from the second equation and three times it from the third equation. We obtain

$$\begin{aligned} x + \frac{2}{3}y + \frac{7}{3}z &= \frac{4}{3} \\ \frac{5}{3}y - \frac{11}{3}z &= \frac{7}{3} \\ 2y - 6z &= 3. \end{aligned} \tag{3}$$

We proceed with the last two equations, now in only two unknowns, as in one earlier example. We divide the second equation of system (3) by $\frac{5}{3}$, obtaining

$$\begin{aligned} x + \frac{2}{3}y + \frac{7}{3}z &= \frac{7}{5} \\ y - \frac{11}{5}z &= \frac{7}{5} \\ 2y - 6z &= 3. \end{aligned} \tag{4}$$

Then we subtract twice this modified second equation from the last equation of (4) yielding

$$\begin{aligned} x + \frac{2}{3}y + \frac{7}{3}z &= \frac{4}{3} \\ y - \frac{11}{5}z &= \frac{7}{5} \\ -\frac{8}{5}z &= \frac{1}{5}. \end{aligned} \tag{5}$$

We divide the last equation by $-8/5$, obtaining for it, $z = -1/8$. Thus, we have replaced our original system of equations (1) by an equivalent system (6).

$$\begin{aligned} x + \frac{2}{3}y + \frac{7}{3}z &= \frac{4}{3} \\ y - \frac{11}{5}z &= \frac{7}{5} \\ z &= -\frac{1}{8} \end{aligned} \qquad (6)$$

Before we work back in system (6) to actually solve for x, y, and z, let's consider an important point. In the transformation of system (1) to system (6) our computation really involved only the coefficients and right-hand constants. So we could display the essential information about system (1) in the form of an array of numbers. Thus, for (1) we could write the array

$$\begin{bmatrix} 3 & 2 & 7 & 4 \\ 2 & 3 & 1 & 5 \\ 3 & 4 & 1 & 7 \end{bmatrix} \qquad (1')$$

We know that certain operations can be performed on systems of equations without changing the solution set of the original equations--this is what we have just done to obtain system (6). These operations can be carried over to operations on the array, such as (1'). Among these operations are the following:

1. We may multiply (or divide) any equation of the system by any constant except zero. This does not change the equation in an essential way. This means that in the array we may multiply (or divide) each number in a row by any constant except zero.
2. We may interchange the order in which the equations are written. This means we may interchange rows of the array.
3. We may add (or subtract) a constant multiple of one equation to (or from) another equation. Thus, for the array we may add (or subtract) a constant multiple of one row to (or from) another row.

The systematic method we followed in going from system (1) to system (6) may be represented in arrays as follows:

$$\begin{bmatrix} 3 & 2 & 7 & 4 \\ 2 & 3 & 1 & 5 \\ 3 & 4 & 1 & 7 \end{bmatrix} \qquad (1')$$

Divide row 1 by 3

$$\begin{bmatrix} 1 & \frac{2}{3} & \frac{7}{3} & \frac{4}{3} \\ 2 & 3 & 1 & 5 \\ 3 & 4 & 1 & 7 \end{bmatrix} \qquad (2')$$

Subtract twice row 1 from row 2 and three times row 1 from row 3,

$$\begin{bmatrix} 1 & \frac{2}{3} & \frac{7}{3} & \frac{4}{3} \\ 0 & \frac{5}{3} & -\frac{11}{3} & \frac{7}{3} \\ 0 & 2 & -6 & 3 \end{bmatrix} \qquad (3')$$

Divide row 2 by $\frac{5}{3}$

$$\begin{bmatrix} 1 & \frac{2}{3} & \frac{7}{3} & \frac{4}{3} \\ 0 & 1 & -\frac{11}{5} & \frac{7}{5} \\ 0 & 2 & -6 & 3 \end{bmatrix} \qquad (4')$$

Subtract twice row 2 from row 3,

$$\begin{bmatrix} 1 & \frac{2}{3} & \frac{7}{3} & \frac{4}{3} \\ 0 & 1 & -\frac{11}{5} & \frac{7}{5} \\ 0 & 0 & -\frac{8}{5} & \frac{1}{5} \end{bmatrix} \qquad (5')$$

Divide row 3 by $-\frac{8}{5}$,

$$\begin{bmatrix} 1 & \frac{2}{3} & \frac{7}{3} & \frac{4}{3} \\ 0 & 1 & -\frac{11}{5} & \frac{7}{5} \\ 0 & 0 & 1 & -\frac{1}{8} \end{bmatrix} \qquad (6')$$

If we write the system of equations corresponding to array (6'), we see that our system is identical with equations (6). Thus, instead of performing our systematic method on the equations as such, we may operate on the array.

Now let's turn back to complete the solution of system (6). We see that $z = -1/8$, and working backwards in this system we find $y = 9/8$ and $x = 7/8$. We often refer to this operation of working back as the "back solution."

Exercise 7-4 Set B

Draw a flow chart to indicate the sequence of calculations in the solution of the following system of equations. Follow the systematic method described in this section.

$$a_{11}x_1 + a_{12}x_2 = b_1$$

$$a_{21}x_1 + a_{22}x_2 = b_2$$

Compare your flow chart with the one in Chapter 6.

7-5 Simultaneous Linear Equations: Gauss Algorithm

A System of 3 Equations

We now want to describe an algorithm, the Gauss Algorithm, for the solution of any system of three simultaneous linear equations in three unknowns. Since we will later want to consider more than three equations, we want to write our equations in a way that we can easily generalize. Thus, instead of using the variables x, y and z, we will use subscripted variables x_1, x_2 and x_3. The coefficients will be identified with two subscripts, one to denote the first, second or third equation and the other to indicate the variable which is being multiplied. For example, a_{12} denotes the coefficient of x_2 in the first equation. Since the right side of each equation is a constant, these right sides are conveniently denoted by quantities with a single subscript. Thus, we consider the equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \qquad (7)$$

Thus, in the last example discussed we have $a_{11} = 3$, $a_{23} = 1$, $a_{32} = 4$, $b_2 = 5$, etc.

Now the essential information about the system of equations is given completely by the coefficients and so it can be displayed clearly in the form of an array of numbers

$$\begin{array}{cccc} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array} \qquad (8)$$

Of course, array (8) corresponds to array (1') in our previous example.

Now let us proceed to describe how to find the solution set of the simultaneous equations (7) represented by array (8). You will recognize that we are following precisely the systematic method we just used in the numerical example. In fact, you will find it helpful to look back at the example after each step you read in the general discussion.

1. We begin by dividing the first equation through by a_{11}. In other words, we divide the numbers of the first row of the array (8) by a_{11}. Thus, we must calculate $a_{11}/a_{11} = 1$, a_{12}/a_{11}, a_{13}/a_{11} and b_1/a_{11}. These numbers are the first row of a new array and it would be convenient to call them a_{11}, a_{12}, a_{13}, and b_1, again, the subscript 1 indicating the first row as usual. But if we try this and start to carry out the calculations in the order indicated, we will replace a_{11} by 1 before we calculate the new a_{12}, a_{13} and b_1. Thus, when we divide by a_{11} we are really dividing by 1 instead of by the old value of a_{11} as we intended to do. One way out of this difficulty is to calculate b_1/a_{11}, a_{13}/a_{11} and a_{12}/a_{11}, the new values of b_1, a_{13} and a_{12} in this order; that is, before we calculate $a_{11}/a_{11} = 1$ the new value of a_{11}. Actually there is no point in calculating a_{11}/a_{11} which we know to be 1. It would be better just to leave a_{11} untouched and remember that the coefficient of x_1 in equation (1) is actually 1 whenever we have need of it. Then there is no need to reverse the order of the calculations. This is what we shall do. The modification of the first row, which we will refer to as "normalizing", is indicated in the flow chart fragment of Figure 7-24.

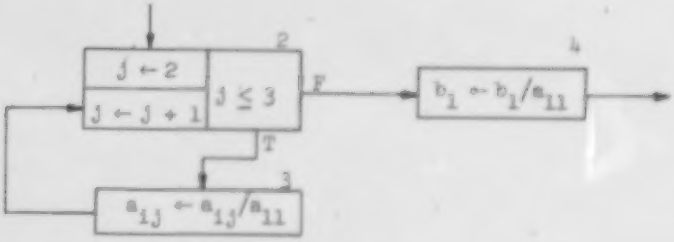

2
j ← 2
j ← j + 1
j ≤ 3
F
T
4
$b_1 \leftarrow b_1/a_{11}$
3
$a_{1j} \leftarrow a_{1j}/a_{11}$

Figure 7-24. Normalizing the first row

The use of the iteration box here may seem a little silly. We present the calculations in this form for later generalization to the case of n equations in n unknowns. What would you have to change in the loop shown in this flow chart to make it apply to 4 equations? n equations?

2. The next step is the elimination of x_1 from both the second and third equations. We need to subtract a_{21} times the new first equation from the second equation and a_{31} times the new first equation from the third equation. The new elements of the second row will be

$$a_{21} - a_{21} \times 1 = 0, \quad a_{22} - a_{21}a_{12}, \quad a_{23} - a_{21}a_{13}, \quad b_2 - a_{21}b_1$$

Again we want to use the same names a_{21}, a_{22}, a_{23}, b_2 for the new elements of the second row. We have the same problem as earlier if we calculate a_{21} first and set it to zero. We again avoid the difficulty by leaving a_{21} untouched and remembering that the coefficient of the corresponding equation is really zero. The calculation of the new elements of the second and third rows is indicated in the flow chart of Figure 7-25.

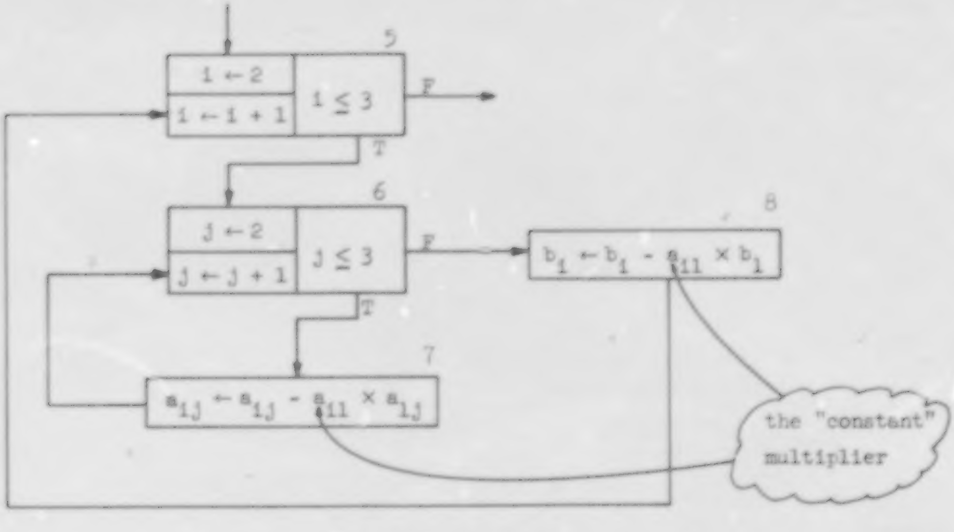

5
i ← 2
i ← i + 1
i ≤ 3
F
T
6
j ← 2
j ← j + 1
j ≤ 3
F
T
8
$b_i \leftarrow b_i - a_{i1} \times b_1$
7
$a_{ij} \leftarrow a_{ij} - a_{i1} \times a_{1j}$
the "constant" multiplier

Figure 7-25. Subtracting suitable multiples of the first row from all the following rows

We see that the variable i controls the row on which we are working, while j determines the entry in the i^{th} row currently receiving the treatment. Observe that the "constant" multiplier a_{i1} (meaning independent of j) is used in computing every element in the new array except for those in the first row and those in the first column.

What changes would be necessary if there were 4 equations in 4 unknowns? n equations in n unknowns? With these changes made, how many times would the outer loop of Figure 7-25 be executed with 4 equations in 4 unknowns? the inner loop? Answer these same questions for n equations in n unknowns.

3. Now we are ready to proceed with the elimination of x_2 from the third equation. But, let us first take a look at our array as it now stands.

$$\begin{array}{c|ccc} \multicolumn{1}{c}{\textcircled{1}} & a_{12} & a_{13} & b_1 \\ \cline{2-4} \textcircled{0} & a_{22} & a_{23} & b_2 \\ \textcircled{0} & a_{32} & a_{33} & b_3 \end{array}$$

Figure 7-26. The present state of affairs

The decorated entries in our array represent values which we are keeping in our heads. We have circled the zeros and boxed the 1's. The task of eliminating x_2 from the third equation will employ the methods illustrated in Figures 7-24 and 7-25 except that the methods are applied to the smaller array blocked off in the lower right of Figure 7-20.

First, we divide the second row of Figure 7-26 by a_{22}, remembering that the new a_{22} will be 1. Again, we use the iteration box for the purposes of later generalization.

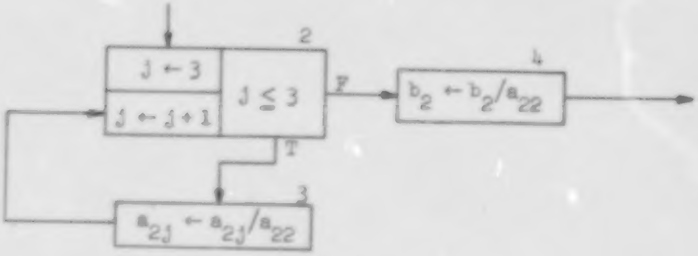

2
$j \leftarrow 3$
$j \leftarrow j + 1$
$j \leq 3$
F
T
4
$b_2 \leftarrow b_2/a_{22}$
3
$a_{2j} \leftarrow a_{2j}/a_{22}$

Figure 7-27. Normalizing the second row

What changes would be necessary in Figure 7-27 if there were 4 equations? n equations? How many times would the loop be executed if there were 4 equations? n equations?

Note that the flow chart of Figure 7-27 which describes the division of the second row to the right of a_{22} by a_{22} is entirely similar to that of Figure 7-24 which describes the division of the first row to the right of a_{11} by a_{11}. We can therefore draw a single flow chart, Figure 7-28, to describe the division of the k^{th} row to the right of a_{kk} by a_{kk}, where we understand that $k = 1$ or 2. Thus, we have

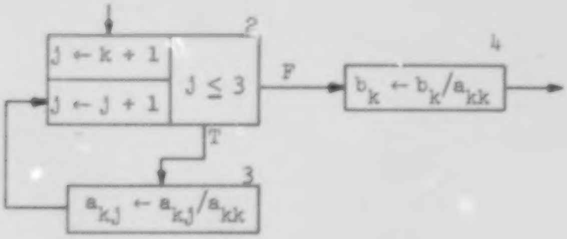


Figure 7-28. Normalizing the k^{th} row

4. Next we eliminate x_2 from the third equation by subtracting a multiple of the second equation from the third equation; that is, we subtract a_{32} times the new second row from the third row of the array. This is indicated by the flow chart in Figure 7-29. This flow chart seems ridiculuous complicated since each of the indexing variables i and j assume only one value before passing out of the loop. Again, our motivation is eventual generalization.

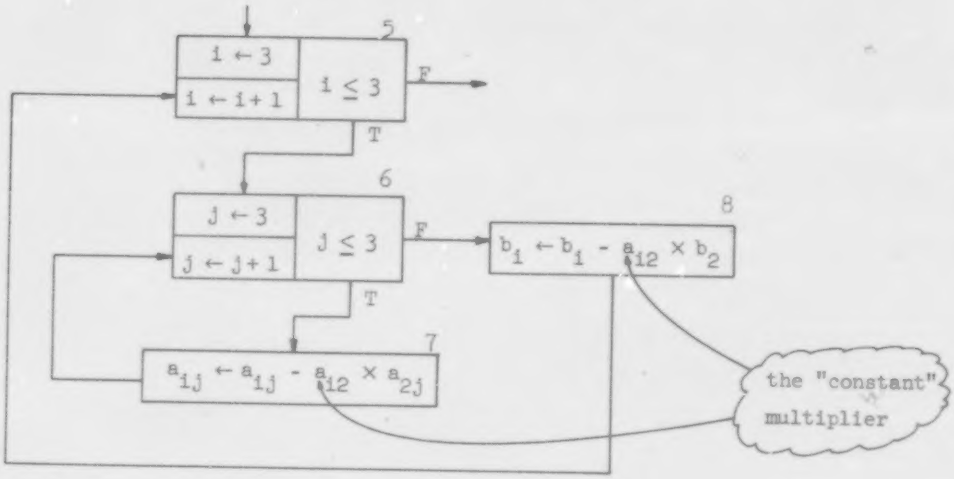

5
i ← 3
i ← i+1
i ≤ 3
F
T
6
j ← 3
j ← j+1
j ≤ 3
F
T
8
b_i ← b_i - a_i2 × b_2
7
a_ij ← a_ij - a_i2 × a_2j
the "constant" multiplier

Figure 7-29. Subtracting suitable multiples of the second row from all the following rows

How many times is the outer loop in this flow chart executed? the inner loop? What changes would be necessary if there were 4 equations? n equations? How many times would the outer and inner loops be executed with 4 equations? n equations?

Now, if we compare Figures 7-25 and 7-29, we again see an opportunity to combine them as shown in Figure 7-30. Here, instead of referring to multiples of row 1 (as in Figure 7-25) or multiples of row 2 (as in Figure 7-29), we speak of multiples of row k.

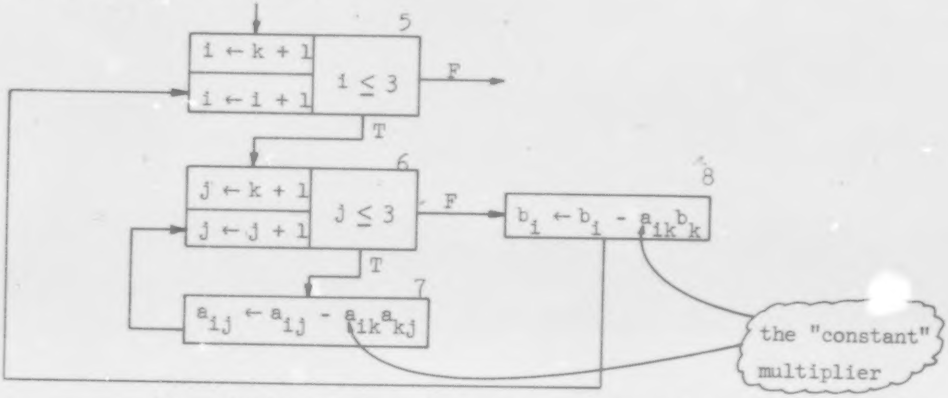

5
i ← k + 1
i ← i + 1
i ≤ 3
F
T
6
j ← k + 1
j ← j + 1
j ≤ 3
F
T
8
b_i ← b_i - a_ik b_k
7
a_ij ← a_ij - a_ik a_kj
the "constant" multiplier

Figure 7-30. Subtracting suitable multiples of the k^{th} row from all following rows

5. Let us take a look at our array after the preceding modifications have been made.

$$\begin{array}{cccc}
\boxed{1} & a_{12} & a_{13} & b_1 \\
\textcircled{0} & \boxed{1} & a_{23} & b_2 \\
\cline{3-4}
\textcircled{0} & \textcircled{0} & \multicolumn{1}{|c}{a_{33}} & b_3
\end{array}$$

Figure 7-31. Penultimate array

We see that all that remains to be done is to divide the third row by a_{33}. This is actually achieved by the flow chart of Figure 7-28 with $k = 3$. In this case, we have to realize that $j \leftarrow k + 1$ means that $j \leftarrow 4$ and, hence, it is never true that $j \leq 3$. Thus, the loop is never executed but instead we go directly to the box in which we have $b_k \leftarrow b_k/a_{kk}$, or since $k = 3$, $b_3 \leftarrow b_3/a_{33}$. When this has been done, our array will have its final form shown in Figure 7-32(a).

$$\begin{array}{cccc}
\boxed{1} & a_{12} & a_{13} & b_1 \\
\textcircled{0} & \boxed{1} & a_{23} & b_2 \\
\textcircled{0} & \textcircled{0} & \boxed{1} & b_3
\end{array}
\qquad\qquad
\begin{array}{r}
x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \\
x_2 + a_{23}x_3 = b_2 \\
x_3 = b_3
\end{array}$$

(a) Final array (b) System of equations represented by (a)

Figure 7-32

We remind you again that the framed entries in the array of Figure 7-32(a) are the ones we are carrying in our head. The system of equations of Figure 7-32(b) is the system represented by the array to the left of it. We recall that the system of equations of Figure 7-32(b) is equivalent to the system we started out with, the system (7) on the first page of this section. By this we mean that the solutions of the two systems are the same. Furthermore, the final system is trivial to solve. Of course, the values of a_{12}, a_{13}, a_{23}, b_1, b_2 and b_3 of Figure 7-26 are new values and not the values represented by these same variables in (8).

We have obtained the array in Figure 7-32(b) from the array (7) in the following way. For $k = 1, 2$ and 3 we have successively applied the flow chart of Figure 7-28 followed by that of Figure 7-30. Now we want to collect all this in one flow chart. We do this in Figure 7-33.

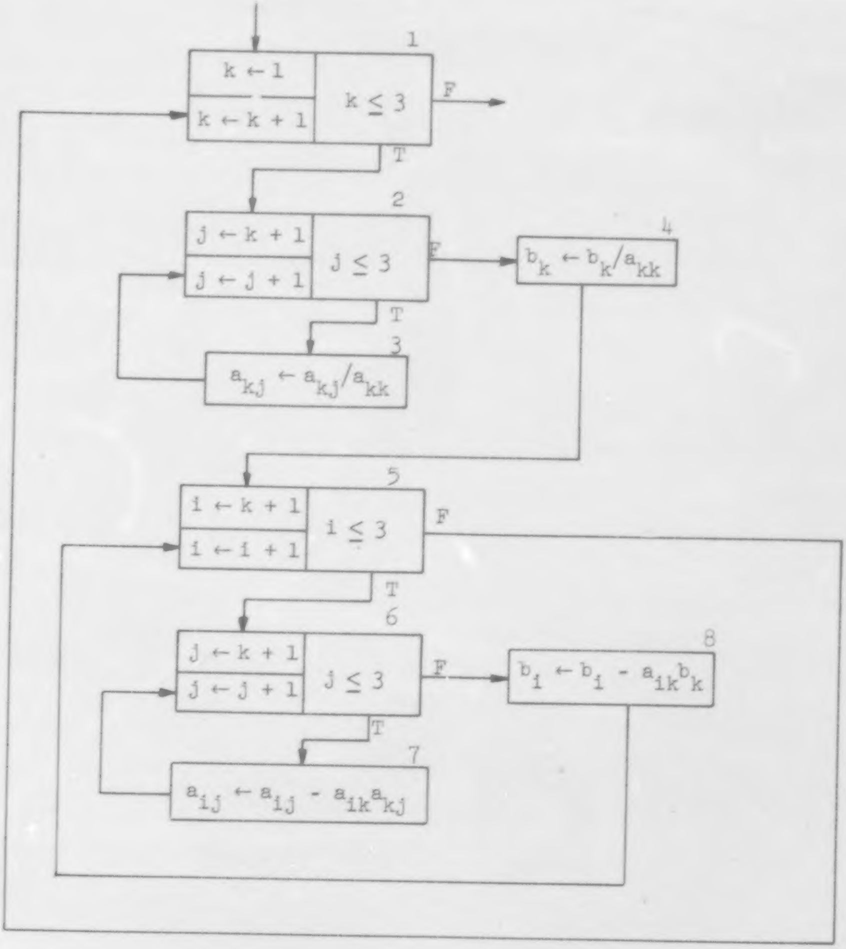

1
k ← 1
k ← k + 1
k ≤ 3
F
T
2
j ← k + 1
j ← j + 1
j ≤ 3
F
T
3
a_kj ← a_kj/a_kk
4
b_k ← b_k/a_kk
5
i ← k + 1
i ← i + 1
i ≤ 3
F
T
6
j ← k + 1
j ← j + 1
j ≤ 3
F
T
7
a_ij ← a_ij - a_ik a_kj
8
b_i ← b_i - a_ik b_k

Figure 7-33. Flow chart for bringing array into equivalent "super-diagonal" form

It is easily checked that Figure 7-33 is obtained by linking the beginning of Figure 7-30 to the end of Figure 7-28 and by hanging the new grouping from an iteration box (box 1) that describes the range for values of k.

The Back Solution. Our algorithm at this stage brings the original system of equations into the form of Figure 7-32(b). We next turn to the relatively simple problem of flowcharting the solution of this system. To see how the mathematical problem is solved, rewrite the system of Figure 7-32 in the form:

$$x_1 = b_1 - a_{13}x_3 - a_{12}x_2$$

$$x_2 = b_2 - a_{23}x_3$$

$$x_3 = b_3 \ .$$

Now we work from the bottom to the top; hence, the "back solution". The value of x_3 is obvious from the third equation. Substituting this value in the second equation we obtain the value of x_2. Substituting the computed values of x_3 and x_2 in the first equation we obtain the value of x_1.

Describing this process in slightly different terms will make it obvious how to construct the algorithm. How do we compute an x_i after all the x's with higher subscripts have been computed? We do it in stages. First, we give x_i the value b_i, then we successively subtract away from the partially computed x_i the value of $a_{ij}x_j$, where j starts with 3 and decreases down to, but not including, i. You should verify this description for x_3, x_2 and x_1. Here, then, is the partial flow chart for computing x_i in Figure 7-34.

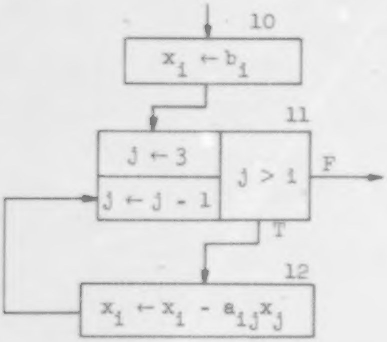


Figure 7-34. Computing the value of x_i

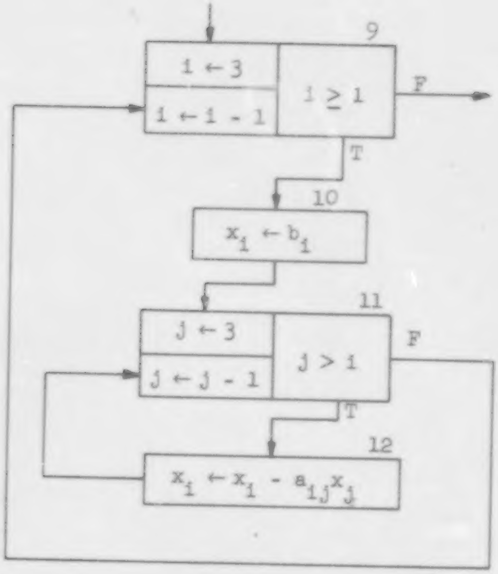

9
$i \leftarrow 3$
$i \leftarrow i - 1$
$i \geq 1$
F
T
10
$x_i \leftarrow b_i$
11
$j \leftarrow 3$
$j \leftarrow j - 1$
$j > i$
F
T
12
$x_i \leftarrow x_i - a_{ij}x_j$

Figure 7-35. The back solution

At last the complete Gaussian algorithm is at hand. The finishing touches merely amount to tacking of the back solution onto the end of Figure 7-33 and providing for the input and output. This is all done in Figure 7-36.

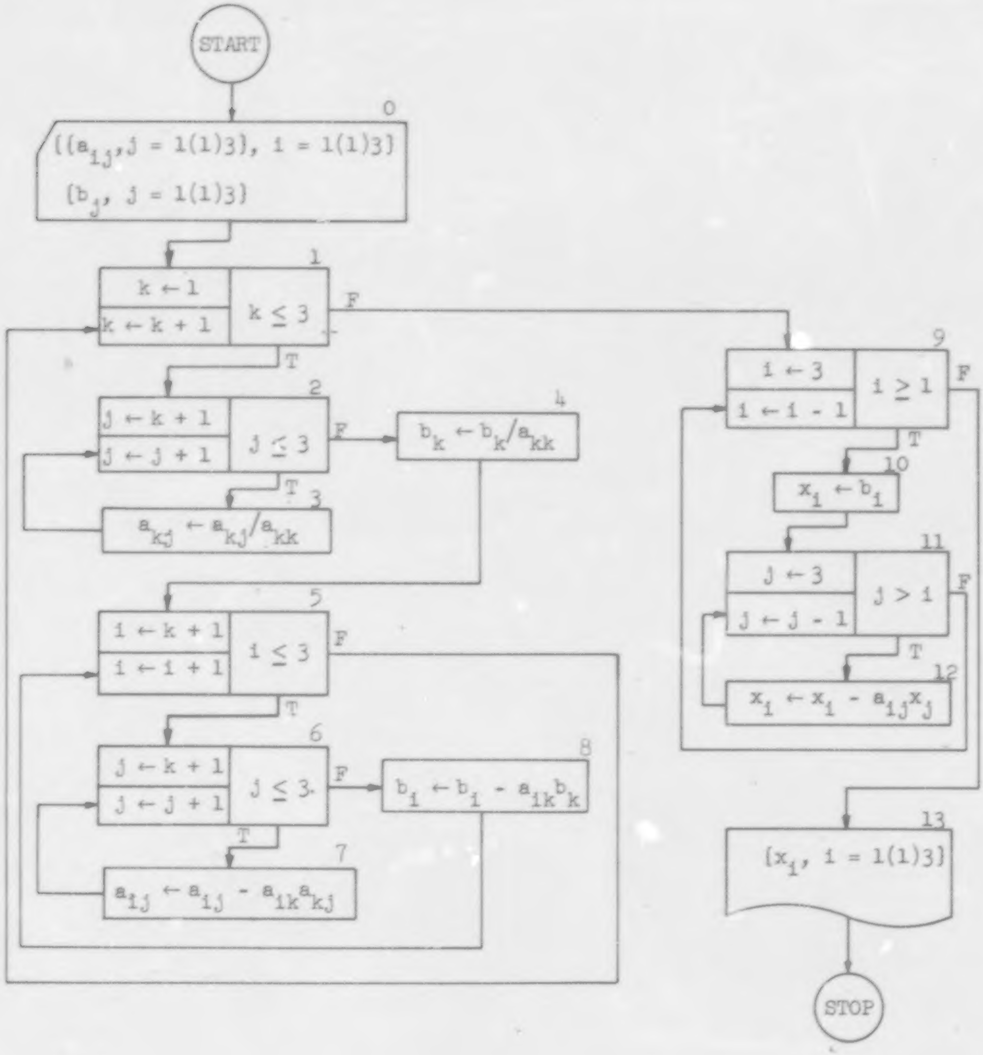

START
0
$\{(a_{ij}, j = 1(1)3\}, i = 1(1)3\}$
$\{b_j, j = 1(1)3\}$
1
$k \leftarrow 1$
$k \leftarrow k + 1$
$k \leq 3$
F
T
2
$j \leftarrow k + 1$
$j \leftarrow j + 1$
$j \leq 3$
F
T
3
$a_{kj} \leftarrow a_{kj}/a_{kk}$
4
$b_k \leftarrow b_k/a_{kk}$
5
$i \leftarrow k + 1$
$i \leftarrow i + 1$
$i \leq 3$
F
T
6
$j \leftarrow k + 1$
$j \leftarrow j + 1$
$j \leq 3$
F
T
7
$a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}$
8
$b_i \leftarrow b_i - a_{ik}b_k$
9
$i \leftarrow 3$
$i \leftarrow i - 1$
$i \geq 1$
F
T
10
$x_i \leftarrow b_i$
11
$j \leftarrow 3$
$j \leftarrow j - 1$
$j > i$
F
T
12
$x_i \leftarrow x_i - a_{ij}x_j$
13
$\{x_i, i = 1(1)3\}$
STOP

Figure 7-36. Complete Gaussian algorithm for 3 equations in 3 unknowns

We have synthesized the Gaussian algorithm very deliberately, step-by-step, so that we do not feel it necessary to analyze its flow chart. Instead, we provide in Figure 7-37 a silhouette of Figure 7-36 with an indication of the purposes of the various loops.

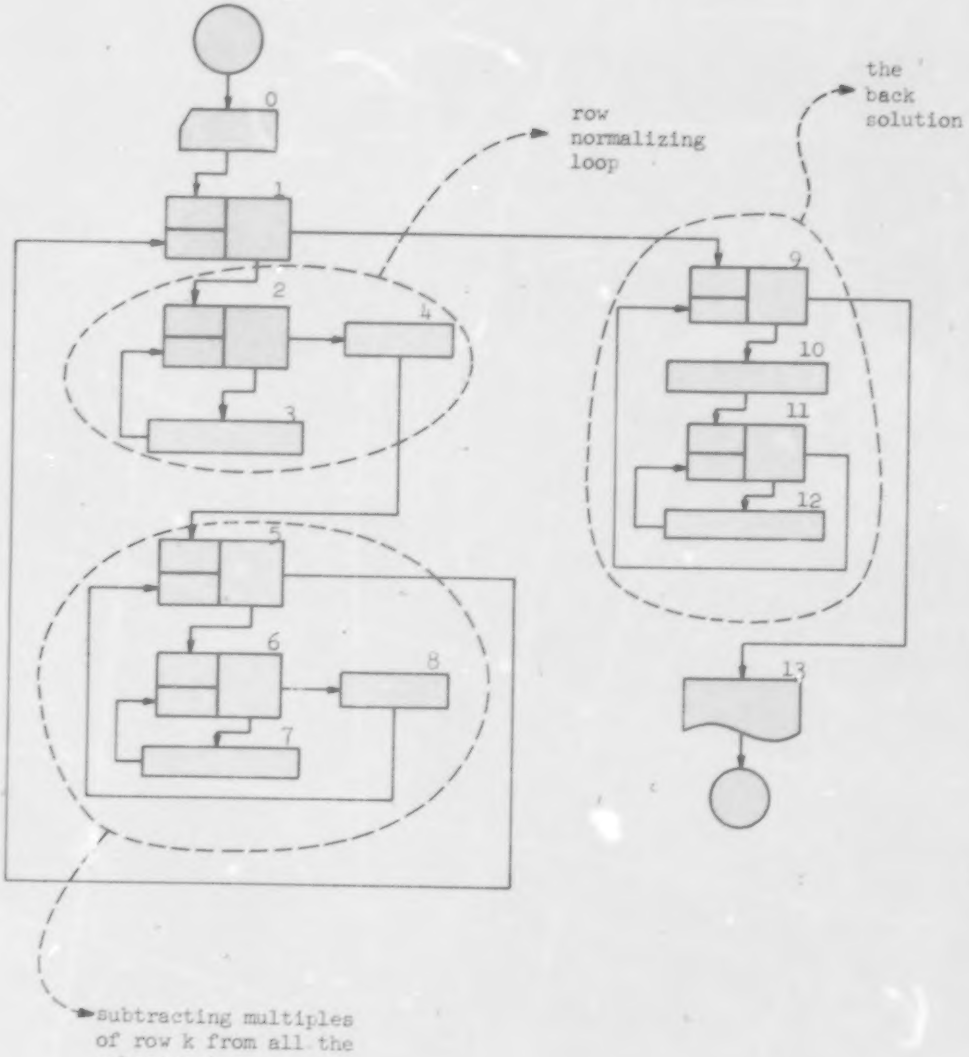

0
1
2
3
4
5
6
7
8
9
10
11
12
13
row normalizing loop
the back solution
subtracting multiples of row k from all the subsequent rows

Figure 7-37. Silhouette of flow chart showing decomposition into several components

Exercises 7-5 Set A

1. Now that you have dutifully followed the preceding discussion, specify the minor changes in the flow chart of Figure 7-34 so as to make it applicable to systems of n equations in n unknowns.

2. Draw the flow chart embodying the changes in Problem 1 and, while you are about it, make the flow chart into a procedure. You must properly handle the modified treatments of input and output and give careful attention to what goes into the funnel. It is not required that you provide alternate exits (even though their necessity should arise) but you may do so if you wish. The system of n equations in n unknowns and the corresponding array are shown below.

$$\begin{array}{l}
a_{11}x_1 + a_{12}x_2 + a_{13}x_n + \dots + a_{1n}x_n = b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \dots + a_{2n}x_n = b_2 \\
- \\
a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \dots + a_{nn}x_n = b_n
\end{array}$$

$$\begin{array}{cccccc}
a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & b_1 \\
a_{21} & a_{22} & a_{23} & \cdots & a_{2n} & b_2 \\
- - & - - & - - & - - & - - & - - \\
a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} & b_n
\end{array}$$

Tracing the Gaussian Algorithm

We are ready now to carry out an actual example working our way step by step through the flow chart of Figure 7-36. You will need to refer to this flow chart frequently as you read. You will also need paper and pencil to check the calculations at each step. The equations we will solve are:

$$\begin{aligned} 2x - y + 6z &= 3 \\ 3x - 4y + 4z &= 1 \\ x + 2y - 5z &= 7 \end{aligned}$$

We can write our array in the form

$$\begin{array}{cccc} 2 & -1 & 6 & 3 \\ 3 & -4 & 4 & 1 \\ 1 & 2 & -5 & 7 \end{array}$$

The method of solution, you will recall, consists of constructing an equivalent set of equations of the form seen in Figure 7-32(b). The array for this second set has the form

$$\begin{array}{cccc} 1 & w & w & w \\ 0 & 1 & w & w \\ 0 & 0 & 1 & w \end{array}$$

where the w's are what we are to determine. Once we know them, the back solution finishes the problem.

After the coefficients are input in box 0, we enter the iteration box 1 of Figure 7-36 and set $k = 1$. This leads us to the iteration box 2. The loop around boxes 2 and 3 (twice) and out through box 4 divides the numbers in the first row (except for the first) by 2. Recall that we know the first element becomes 1 when we divide it by itself. As explained before, it is easier in machine computation not to actually replace the 2 with a 1. However, we will place a square around the 2 to remind us that the coefficient of x_1 is actually 1. After box 4 and before box 5 the array has become:

$$k = 1 \qquad \begin{bmatrix} \boxed{2} & -\frac{1}{2} & 3 & \frac{3}{2} \\ 3 & -4 & 4 & 1 \\ 1 & 2 & -5 & 7 \end{bmatrix}$$

We enter box 5 and set $i = 2$. This leads to box6 where we set $j = 2$. Now we loop through boxes 6 and 7 twice and out through box 8. This looping accomplishes the subtraction of 3 times the first equation from the second on the array elements in which we are interested. We circle the 3 in the second row because we know the coefficient of x_1 in the second equation is now zero. As we re-enter box 5 for the second time the array has become

$$k = 1 \qquad \begin{bmatrix} \boxed{2} & -\frac{1}{2} & 3 & \frac{3}{2} \\ \textcircled{3} & -\frac{5}{2} & -5 & -\frac{7}{2} \\ 1 & 2 & -5 & 7 \end{bmatrix}$$

Now $i = 3$ in box 5 and we re-enter box 6 to loop to box 7 twice and out box 8. This accomplishes the subtraction of the first equation from the third on the array elements we will use in our solution. We will just circle the 1 in the first position of the third row to remind us that this coefficient of x_1 is now zero. Our matrix has now become

$$k = 1 \qquad \begin{bmatrix} \boxed{2} & -\frac{1}{2} & 3 & \frac{3}{2} \\ \textcircled{3} & -\frac{5}{2} & -5 & -\frac{7}{2} \\ \textcircled{1} & \frac{5}{2} & -8 & \frac{11}{2} \end{bmatrix}$$

We return to box 5 with $i = 4$ and are sent back to box 1 where k becomes 2. Again we loop through boxes 2 and 3 and out through box 4 accomplishing the division of the last two elements of the second row by $-\frac{5}{2}$. As before, we place a square around the $-\frac{5}{2}$ in the second row. The matrix as we leave box 4 for the second time is:

$$k = 2 \qquad \begin{bmatrix} \boxed{2} & -\frac{1}{2} & 3 & \frac{3}{2} \\ \textcircled{3} & \boxed{-\frac{5}{2}} & 2 & \frac{7}{5} \\ \textcircled{1} & \frac{5}{2} & -8 & \frac{11}{2} \end{bmatrix}$$

Now we enter box 5 with $i = 3$. We flow on to box 6 with $j = 3$. The loop from box 6 to 7 and out through box 8 accomplishes the subtraction of $\frac{5}{2}$ times the second equation from the third. Again we circle the $\frac{5}{2}$ in the third row because this coefficient of x_2 is really zero and the matrix now is

$$k = 2 \qquad \begin{bmatrix} \boxed{2} & -\frac{1}{2} & 3 & \frac{3}{2} \\ \textcircled{3} & \boxed{-\frac{5}{2}} & 2 & \frac{7}{5} \\ \textcircled{1} & \textcircled{\frac{5}{2}} & -13 & 2 \end{bmatrix}$$

Back to box 5, i is set equal to 4 and we flow on to box 1 with k becoming 3. Now for the last time we enter box 2 where j fails the test and we go on to box 4. After box 4 the matrix is this:

$$k = 3 \qquad \begin{bmatrix} \boxed{2} & -\frac{1}{2} & 3 & \frac{3}{2} \\ \textcircled{3} & \boxed{-\frac{5}{2}} & 2 & \frac{7}{5} \\ \textcircled{1} & \textcircled{\frac{5}{2}} & \boxed{-13} & -\frac{2}{13} \end{bmatrix}$$

As we flow on to box 5, $i = 4$ and fails the test. Back to box 1, k becomes 4 and fails its test sending us on to box 9 and the back solution. Our array has reached its final form. Henceforth, all assignments are to components of the vector x.

We leave box 9 with $i = 3$ and proceed to box 10 where x_3 receives its initial value

$$x_3 = -\frac{2}{13}.$$

On to box 11 where j is set equal to 3 and fails the test, returning us to box 9 where i is reduced to 2. In box 10, x_2 is initialized with the value

$$x_2 = \frac{7}{5}.$$

Now one transit of the loop of boxes 11 and 12 gives x_2 the new value

$$x_2 = \frac{7}{5} - 2 \times (-\frac{2}{13}) = \frac{111}{65} .$$

The failure of the test on returning to box 11 sends us again to box 9 where i is set to 1. And now, on our last time through box 10, x_1 is initialized

$$x_1 = \frac{3}{2} .$$

Two successful transits of the loop of boxes 11 and 12 with $j = 3$ and $j = 2$ give x_1, succesively, the values

$$x_1 = \frac{3}{2} - 3 \times (-\frac{2}{13}) = \frac{51}{26}$$

and

$$x_1 = \frac{51}{26} - (-\frac{1}{2}) \times \frac{111}{65} = \frac{366}{130} = \frac{183}{65} .$$

When j is set to 1 on the last time into box 11, the test is failed and we proceed to box 9 where i is set to 0 and fails the test. On to the finish line! In box 13 we are instructed to print out the present values of x_1, x_2 and x_3, namely,

$$\frac{183}{65}, \frac{111}{65}, -\frac{2}{13} .$$

__Exercises__ __7-5__ __Set__ __B__

1. Determine whether the values computed for x_1, x_2 and x_3 actually satisfy the equations.

2. Make a trace of the flow chart of Figure 7-36 similar to the one just concluded in the text for the system:

$$3x + 4y + z = -7$$

$$2x + 4y + z = 3$$

$$3x - 5y + 3z = 7.$$

List the intermediate arrays at the same points, as done in the text.

3. Using the flow chart you developed in Problem 2 of Exercises 7-5, Set A, for the solution of n equations in n unknowns, make a trace for the system of equations:

$$\begin{aligned} 3x_1 - 2x_2 + 7x_3 - x_4 &= 2 \\ 2x_1 + 3x_2 - 4x_3 + x_4 &= 7 \\ x_1 + 2x_2 + 5x_3 + 2x_4 &= 11 \\ 4x_1 + 3x_2 + 7x_3 - 8x_4 &= -2 \end{aligned}$$

Compute the entries of your sequence of arrays in decimal form using a three-digit chop. [Warning: This problem is extremely tedious. It should give you the proper respect for a computer which can solve, say, 100 equations in 100 unknowns.]

Partial pivoting

We have presented in Figure 7-36 the Gaussian algorithm for the solution of 3 equations in 3 unknowns. And you, in Problem 2 of Exercises 7-5, Set A, adapted this algorithm to a procedure for solving n equations in n unknowns. In both these flow charts a difficulty may arise which we have not yet considered.

In the normalizing part of the process (see Figures 7-37 and 7-36) we divided the k^{th} row through by a_{kk} which we call the pivot element. It could very well happen that at some stage $a_{kk} = 0$. What are we to do in such a case? Indeed, if $a_{kk} \neq 0$ but is very small, then we are dividing by a small quantity and we shall obtain a large result and we may very well magnify any errors which are creeping into the solution. We gave an example of what can happen in such problems in Chapter 6. There we considered the system of equations

$$\begin{aligned} .0001x + y &= 1 \\ x + y &= 2 \end{aligned} \qquad (13)$$

We saw that if we could use exact arithmetic we could get the exact solution. But in all computing machines we are restricted to a small finite number of digits and so most numbers can be represented only approximately. For example, if we are restricted to 3-digit arithmetic, we saw in Chapter 6 that the order of the equations used in elimination was most important. When we divided the first equation of the system (13) by .0001 we obtained

$$x + 10000y = 10000.$$

The elimination of x from the second equation gave

$$-9990y = -9990$$

since only 3 significant digits could be retained and the results were chopped. Then $y = 1$. Substitution into the first equation yields $x = 0$. Actually, the true solution obtained using exact arithmetic is

$$x = 10000/9999, \quad y = 9998/9999.$$

We have obtained a very poor approximation to the solution.

On the other hand, when we took the equations in the order

$$x + y = 2$$
$$.0001x + y = 1$$

we obtained for the second equation

$$.999y = .999 \quad .$$

Again we find $y = 1$; substitution into the first equation now gives $x = 1$. We have this time obtained a good approximation to the solution.

Thus, one way of solving the system gives very inaccurate results, whereas another method gives better results. It appears that in order to avoid difficulties of division by small quantities we should try to choose our divisor to be as large as possible. This can be achieved by interchanging the order in which the equations are written down. Of course, the order in which the equations are written has no effect on the true mathematical solution. On the other hand, we have just seen that a change in the order may give better computer results. Consequently, we should be alert, at all stages of our algorithm, for a change in the order which might produce better results. Interchanging the order of our equations clearly corresponds to interchanging the rows of our arrays.

We will explain how this works with an example.

$$\begin{array}{cccccc}
\boxed{1} & 3 & -15 & 8 & 4 & -9 \\
\textcircled{0} & \boxed{1} & 2 & 0 & 3 & 2 \\
\cline{3-6}
\textcircled{0} & \textcircled{0} & \multicolumn{1}{|c}{\frac{1}{2}} & 3 & 4 & 6 \\
\textcircled{0} & \textcircled{0} & \multicolumn{1}{|c}{4} & 2 & 9 & 14 \\
\textcircled{0} & \textcircled{0} & \multicolumn{1}{|c}{-11} & 6 & 2 & -3
\end{array}$$

Figure 7-38. In the midst of solving 5 equations in 5 unknowns

In Figure 7-38 we are in the midst of solving 5 equations in 5 unknowns. We next want to eliminate the variable x_3 from the fourth and fifth equations.

Our next step normally would be to normalize the third row. Then we would subtract suitable multiples of the normalized row from the fourth and fifth rows.

But, first we look to see whether we can make a row interchange so as to get a larger pivot element. We look down the third column for the element of largest absolute value. But, we only look in the blocked off portion of the first column as the first two equations have already been processed and will not be used again until the back solution. The three entries which thus come under consideration are $\frac{1}{2}$, 4 and -11. Of these, -11 has the largest absolute value, so we interchange the third and fifth rows of our array, as seen in Figure 7-39:

$$\begin{array}{cccccc}
\boxed{1} & 3 & -15 & 8 & 4 & -9 \\
\textcircled{0} & \boxed{1} & 2 & 0 & 3 & 2 \\
\cline{3-6}
\textcircled{0} & \textcircled{0} & \multicolumn{1}{|c}{-11} & 6 & 2 & -3 \\
\textcircled{0} & \textcircled{0} & \multicolumn{1}{|c}{4} & 2 & 9 & 14 \\
\textcircled{0} & \textcircled{0} & \multicolumn{1}{|c}{\frac{1}{2}} & 3 & 4 & 6
\end{array}$$

Figure 7-39. After interchanging rows 3 and 5

And now we proceed to normalize using -11 as our pivot element and to eliminate x_3 from the fourth and fifth rows. The process that we have interpolated here is called "partial pivoting".

*Exercise 7-5 Set C

1. Your task here is to prepare a flow chart component for partial pivoting and to insert it in your flow chart (prepared in Exercise 7-5, Set A, Problem 2) for the Gaussian algorithm for n equations in n unknowns. We remind you that the pivoting process consists of:

   (i) searching the k^{th} column from the k^{th} row downward for the element that yields the largest absolute value;

   (ii) interchanging the row in which this largest element occurs with the k^{th} row.

   In the event that your search in (i) yields zero for the maximum absolute value, make a special exit for this case. The main flow chart which calls for the procedure to be executed will then want to print out the message, "system of equations is singular". Introduce into your flow chart the special notation $\boxed{c \longleftrightarrow d}$ as a shorthand for

$$\boxed{\begin{array}{c} \text{copy} \leftarrow c \\ c \leftarrow d \\ d \leftarrow \text{copy} \end{array}} \quad .$$

2. Another method for improving the Gaussian algorithm is called "equilibration". Its purpose is to bring all the equations to approximately the same scale. This is done by multiplying each row of your array by the appropriate positive or negative power of two. This appropriate power of two should be so chosen that the largest of the magnitudes of the $a_{i,j}$'s in that row (not b_i) will be greater than $\frac{1}{2}$ but not greater than 1. [Powers of 2 are used so that in a machine which does its arithmetic operations in binary no round error will be introduced.]

   Prepare a procedure for carrying out equilibration. Also, prepare a master flow chart which will call first for equilibration and then for the improved Gaussian algorithm of Problem 1. The main flow chart should print out the message "system of equations is singular" in the case that some row has all its $a_{i,j}$'s equal to zero.

Chapter 8

COMPILATION AND SOME OTHER NON-NUMERIC PROBLEMS

8-1 *Introduction*

In Chapter 7 we were concerned with problems that lend themselves to numerical solution. We also know, from having studied procedures for manipulating character strings (Section 5-6), that the digital computer is potentially useful in solving another class of problems, i.e., those for which the significant variables and operations are other than arithmetic.

There are, in fact, a large number of interesting problems in this category. For convenience we call these "non-numeric problems" to suggest that, for these, the numerical computation is secondary to the symbol manipulation. We list here three important examples in this category, although only the last of these will be considered here.

(1) *Simulation problems* - where the object is to have the computer imitate an actual process. With simulation, for example, a company can determine how a potential product would behave under all kinds of conditions without going to the trouble and expense of building prototypes.

(2) *Information retrieval* - where the object is to find the set of items, satisfying certain descriptive criteria, in a large well-described set of items. Examples could be finding in the libraries of your state all books describing mastadons and written by Russians, or finding in the armed forces all left-handed, blue-eyed males speaking Swahili.

(3) *Translating a computer program* written in a procedural language similar to FORTRAN or ALGOL into an executable set of machine language instructions.

The translation problem should be of special interest to us. The programs which we have been running on the computer are, of course, sequences of statements which in turn are nothing more than strings of characters. Somehow the computer has been managing to accept our programs as *input data*, analyze them (i.e., as strings of characters) and then come up with other strings of characters (sequences of machine instructions) as the result. Until now we have assumed that this process is too complicated to try to understand. So we have essentially accepted the compiler as a "black box", taking it for granted, happy to have the service it renders. In this chapter we shall take a more detailed view of this black box. Your first step should be to review what we've already learned about manipulation of character strings in Section 5-6. Next, in Section 8-2, we shall develop a few new "tools" which are of use for symbol manipulation, in general, and for language translation, in particular. Then we shall tackle our main problem with proper preparation. In Section 8-3 we describe a simple but representative programming language for which the compiler is to be discussed.

Compilation of a source program to a target program can be thought of as consisting of four parts:

1. Prescan;
2. decomposition of assignment statements to an intermediate form resembling machine language instructions;
3. translation of all other types of statements to an intermediate form also resembling machine language instructions; and
4. translation from intermediate forms to the target language.

Since the language statement is either an assignment or is not, the second and third parts of this description can be thought of as occurring on an "either-or" basis. Many, but not all, of the operations normally associated with the prescan part will be discussed in Section 8-4. The decomposition of assignment statements will be treated in Sections 8-5 and 8-6. The translation of other types of statements (step 3) is not taken up in this text. The translation from intermediate form to target language for assignment statements is not considered in detail because step 4 will be different for each different machine.

8-2 Symbol manipulation

In Section 5-6 we developed a procedure (chekst) which searched for the occurrence of a given substring in a given string. If found, chekst reports the position of the first character of this substring. You may have wondered of what use such a procedure could be.

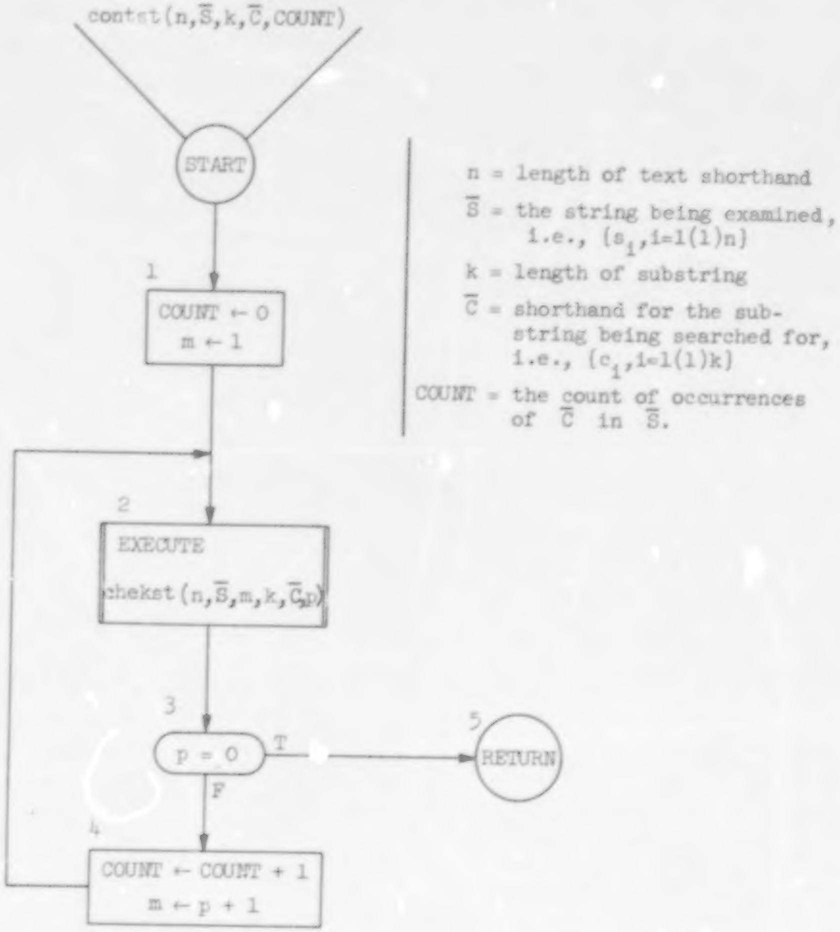
The authorship of some pieces of literature is in doubt; perhaps the piece was published anonymously, perhaps records proving authorship have been lost or destroyed, or perhaps the literature has been discovered unpublished. Often scholars are able to identify the author beyond reasonable doubt, but in some instances the study of scholars has not been able to resolve the question of authorship. In recent years it has been found that, when all else fails, subtle matters involving the choice of words and of forms of words can help in establishing authorship. It is not our purpose to go into the question of how this is actually carried out, rather we would point out that this type of analysis requires such extensive study of the frequency of occurrence of words and phrases that computer processing is required.

Example 1

Prepare a flow chart to count the occurrences of each of a given set of English words in a character string $\overline{S} = \{s_i, i = 1(1)n\}$ which represents a work of literature.

Our first temptation might be to use the chekst procedure that has been mentioned. However, you will recall that another useful procedure was suggested (Section 5-6, Exercise 3) for counting the number of occurrences of a substring in a given string. A flow chart for this procedure, called contst, is given in Figure 8-1. The object of contst is to locate and count occurrences of one substring $\overline{C} = \{c_i, i = 1(1)k\}$ in the string $\overline{S} = \{s_i, i = 1(1)n\}$.

If we think of each word in the given set of English words as a substring, we can solve our problem by repeatedly calling on contst, once for each word in the set. This use of contst is shown in Figure 8-2.


contst(n,$\overline{S}$,k,$\overline{C}$,COUNT)
START
1
COUNT ← 0
m ← 1
2
EXECUTE
chekst(n,$\overline{S}$,m,k,$\overline{C}$,p)
3
p = 0
T
F
5
RETURN
4
COUNT ← COUNT + 1
m ← p + 1

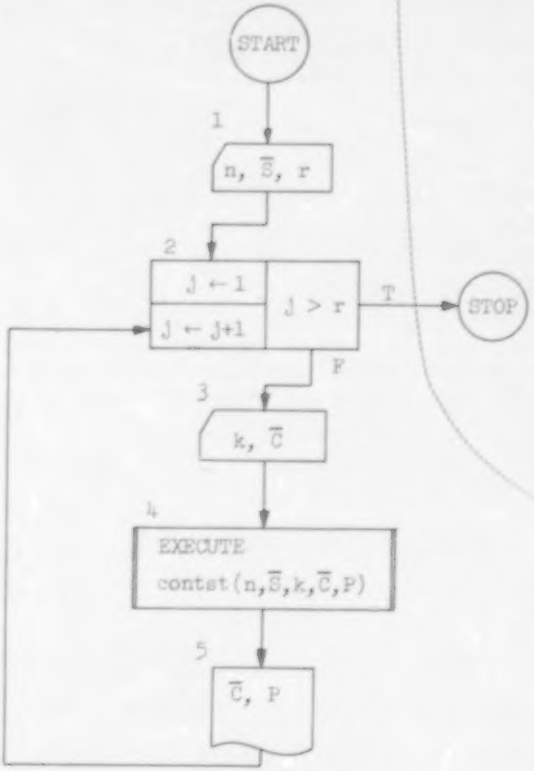
n = length of text shorthand
$\overline{S}$ = the string being examined, i.e., $\{s_i, i=1(1)n\}$
k = length of substring
$\overline{C}$ = shorthand for the substring being searched for, i.e., $\{c_i, i=1(1)k\}$
COUNT = the count of occurrences of $\overline{C}$ in $\overline{S}$.

Figure 8-1. Flow chart of the contst procedure

Notice that the procedure flow chart for contst depends on the chekst flow chart (Section 5-6). Since we now deal extensively with strings of characters, we adopt the shorthand notation $\overline{S}$ standing for the more explicit $\{s_i, i = 1(1)n\}$ which we introduced in Section 5-6. We can refer to a string as a whole (by $\overline{S}$), or to individual elements of a string (as, for example, s_m), or if need be, to substrings (as $\{s_i, i = m(1)n\}$). Third, even with this shorthand, lists of parameters for our procedures will be pretty long. To help minimize the confusion, we will tabulate the key symbols and their explanations on each flow chart.

The work of literature to be scanned is the character string $\overline{S}$. Each English word, the occurrences of which we will count, is to be $\overline{C}$ having a length k. In the main flow chart, Figure 8-2, we let r be the number of English words for which counts of occurrences are required.

First, we input the text and the number of English words for which counts are desired. Next, we read in an English word and count and print its number of occurrences. This is done in a loop. The loop terminates after the counts for r different English words are completed.


START
1
n, $\overline{S}$, r
2
J ← 1
J ← J+1
J > r
T
STOP
F
3
k, $\overline{C}$
4
EXECUTE
contst(n,$\overline{S}$,k,$\overline{C}$,P)
5
$\overline{C}$, P

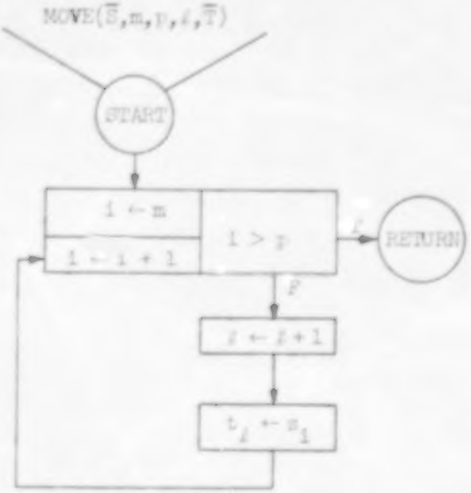
n = length of text
$\overline{S}$ = the string of text
r = number of English words
k = length of an English word
$\overline{C}$ = an English word
p = count of occurrences of $\overline{C}$ in $\overline{S}$.

Figure 8-2. Flow chart to count the occurrences of a set of r words in a text of n characters

What other basic operations might we want to be able to perform on strings? By analogy with the processing of numeric information we can expect that the simple *copying of information*, that is, moving it from one place to another, would be basic to our facility for handling strings of characters.

Example 2

We will design a general *move procedure* to append the elements of $\overline{S}$ between s_m and s_p to a "target" string $\overline{T}$ of length ℓ. A self-explanatory flow chart is given in Figure 8-3.


MOVE($\overline{S}$,m,p,ℓ,$\overline{T}$)
START
i ← m
i ← i + 1
i > p
T
RETURN
F
$\ell \leftarrow \ell + 1$
$t_\ell \leftarrow s_i$

$\overline{S}$ = original string

m = index of first element of $\overline{S}$ to be moved

p = index of last element of $\overline{S}$ to be moved

ℓ = length of target string

$\overline{T}$ = target string

i = pointer for string $\overline{S}$

Figure 8-3. Flow chart of move procedure

In the processing of text we often want to remove certain substrings, say those that represent the so-called interstitial words, such as "the, a, an, of", etc. A special case frequently encountered is to remove a particular single character that may be undesirable in the source string. One way to do this is by moving the parts of the substring we want to keep, skipping over the interstitial words or characters we want to remove. We can perform this deletion with a procedure similar to the move procedure.

Example 3

Develop a flow chart for a procedure called delete with arguments (K, n, $\overline{S}$, m, p, ℓ, $\overline{T}$, $\underline{y}$). Delete appends the m^{th} to the p^{th} elements (inclusive) of the string $\overline{S}$ which has a length n, to the end of the string $\overline{T}$ of length ℓ. During the transfer delete inspects each character for equality to the character K, omitting the character, if equal. Finally, give ℓ the new length of T.

Operation of the delete procedure is pictured below:

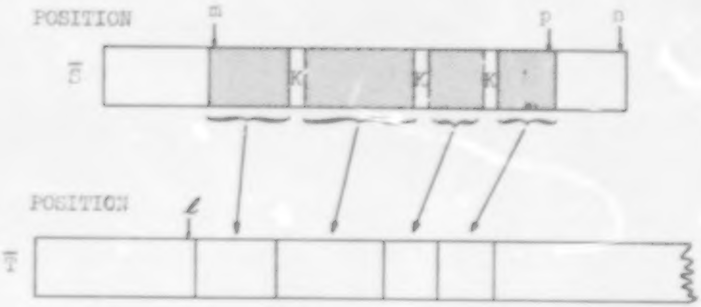
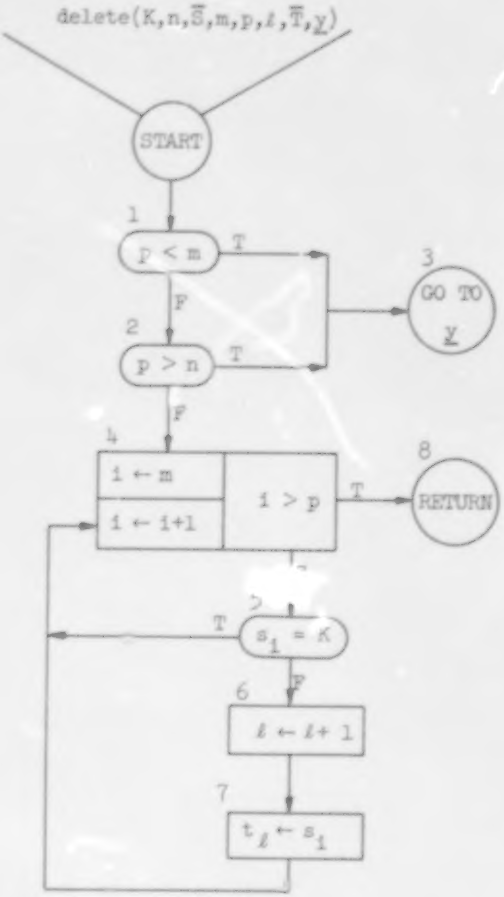


Figure 8-4. Information transfer of delete procedure

The technique is to examine each character of $\overline{S}$ from s_m to s_p, moving it to $\overline{T}$ or not moving it, depending on whether the character being examined is or is not K. Two pointers are needed, one starting at s_m and moving to s_p; the other starting at $t_{\ell+1}$ and advancing so as to point to the next available position of that string.

The flow chart in Figure 8-5 includes an alternate return. If the input parameters are illegal (i.e., $p < m$ or $p > n$), control is transferred to a box label in the flow chart that calls on delete. The box label is represented by $\underline{y}$. The statement of the problem does not call for this provision of an alternate return. It has been included in the solution to suggest the good practice of (whenever possible) verifying the logical validity of input parameters to procedures. The delete procedure will be used later in the text where the use of the alternate return will be shown.


delete$(K,n,\overline{S},m,p,\ell,\overline{T},\underline{y})$
START
1
$p < m$
T
F
3
GO TO $\underline{y}$
2
$p > n$
T
F
4
$i \leftarrow m$
$i \leftarrow i+1$
$i > p$
T
8
RETURN
F
5
T
$s_i = K$
F
6
$\ell \leftarrow \ell + 1$
7
$t_\ell \leftarrow s_i$

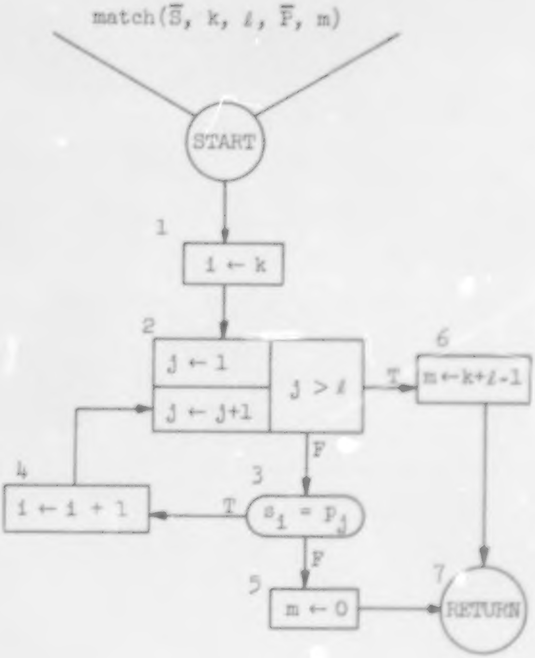
K = character to be deleted
n = length of original string
$\overline{S}$ = source string
m = index of first element of $\overline{S}$ to be examined
p = index of last element of $\overline{S}$ to be examined
ℓ = length of target string
$\overline{T}$ = target string
$\underline{y}$ = a box in the flow chart that calls on delete
i = pointer for string $\overline{S}$

Figure 8-5. Flow chart of delete procedure

In our next example we look at a useful procedure that is similar to chekst but more limited. It determines whether a given substring appears starting with a specific element of a string, $\overline{S}$.

Example 4

Develop the flow chart for a procedure called match that tests the equality of a substring of $\overline{S}$ beginning at s_k with a string $\overline{P}$ of length ℓ. If equality exists, set a pointer m to the index of the rightmost matching element in $\overline{S}$; otherwise, set m to zero. Figure 8-6 shows the flow chart.



$\overline{S}$ = the string to be inspected

k = index of the element of $\overline{S}$ at which inspection is to start

ℓ = length of the substring to be compared with part of $\overline{S}$

$\overline{P}$ = the substring to be compared with part of $\overline{S}$

m = pointer giving the result of the match procedure

i = local (auxiliary variable)

j = local (auxiliary variable)

Figure 8-6. A flow chart of the match procedure

Exercises 8-2

Assume that a character string $\overline{S}$ of length n is in a computer's memory. Draw flow charts to solve Problems 1 - 8. In each case your flow chart should call on one or more of the procedures developed in Section 5-6 or Section 8-2 that seems applicable.

1. Determine if the character "A" occurs at any place after a "B". If so, return a pointer to "A" (set p = the index of "A").

2. Determine if the substring "TH" occurs at any place after the substring "DR". If so, return a pointer to "DR".

3. Determine if the characters "A", "B", and "C" occur in that order, not necessarily adjacently. If so, return a pointer to "B".

4. Identify the most frequently occurring letter in the string.

5. Find out if any letters of the alphabet occur exactly three times and identify them.

6. If "B" immediately follows "A", substitute "X" for each such "B" in the string. Report the count of such substitutions.

7. If the character "A" occurs after the character "Z", remove all intervening commas ",".

8. If the string begins with the substring "NOW", remove that substring from $\overline{S}$.

9. Use the move procedure to modify the flow chart for the delete procedure (Figure 8-5). Comment on the advantages or disadvantages of your result as compared to Figure 8-5.

10. The move and match procedures shown in the text omit the length, n, of the string $\overline{S}$ from their parameter list. Why is this possible in these cases? Can the same omission be made for other flow charts in this section? Which ones? Do you think that this omission would usually be a good idea? Why?

8-3 A language to be translated

Translation of a program from procedural to machine language is actually a form of symbol manipulation. The process is complex, even for simple procedural languages. A compiler for a language having many different statement types, great flexibility in writing expressions and/or several ways to define and call functions, proves to be a rather complex program. Still, the basic ideas behind translation from a procedural language to a machine language are not overly complicated and will be demonstrated for a simplified "make believe" language called TYPICAL. TYPICAL is based on our flow chart language. It is a very simple language but is surprisingly representative of most procedural languages.

Description of TYPICAL

Card format

1. All *statements* will be punched on cards in "free form". Blank columns will be ignored and a statement may begin and end at any card column.
2. Alphabetic literals (i.e., strings of characters between quotation marks) will not be allowed.
3. A semicolon (;) *must* appear after every statement.
4. A statement label *must* be followed by a colon (:).
5. All statement labels and variables (collectively called *identifiers*) must begin with an alphabetic character which may be followed by any collection of alphabetic characters and/or digits (but *no* special characters).

Examples of assignment statements are shown in Figure 8-7.

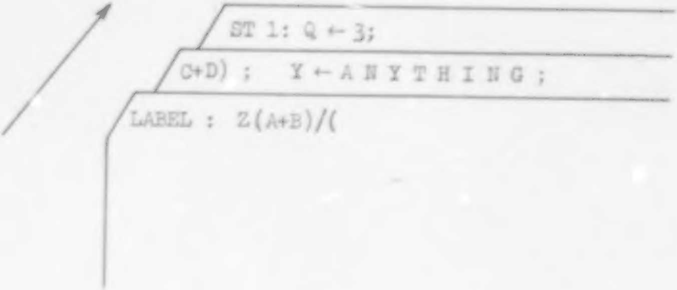


Figure 8-7. Examples of the appearance of assignment statements on cards (The arrow indicates the order of the cards.)

Notice that statements can extend from one card to another or several statements can be on the one card. Note also that the blank spaces may appear anywhere including at the beginning and end of cards and even in the middle of identifiers.

The assignment statement: An assignment is essentially whatever has been written in an assignment box together with a statement label, if needed. For our purposes some specific assumptions and restrictions will be made:

1. all arithmetic is done with "real" numbers,
2. special symbols "←" for assignment and "↑" for exponentiation will be available,
3. arithmetic operators "+, -, /, ×, ↑" and the use of parentheses are allowed,
4. subscripts are *not* allowed,
5. signed variables such as (-Z) are *not* allowed,
6. function references such as $\sqrt{\ }$ or sin are *not* allowed.

Examples of assignment statements permissible in TYPICAL are:

```
Z ← (A + (B × (C - D↑2)));
CALCULATE: AVERAGE ← (X + Y)/2;
LABEL 85:  Z3A ← 85;      etc.
```

The conditional statement: The silhouette of flow chart boxes has been used to convey information about the kind of statement placed in the box. For the assignment statement this information is implicit because of the presence of the left-pointing arrow. For other statements this information must be transmitted via an appropriate descriptive or key word. We choose the following form for the conditional statement:

```
IF (relation) THEN label;
```

For example,

```
IF (A > 0) THEN CALCSQRT;
TEST:  IF (I = J + 7) THEN LABEL 85;
TESTB:  IF (B = 0) THEN DONTDIV; etc.
```

The key word "IF" is used to identify the type of statement.[†] The relation must be enclosed in parentheses. Any simple relation may be used involving one of the six relational symbols ($>$, $\geq$, $=$, $\neq$, $<$, $\leq$). The word "THEN" is added simply for readability and the label preceding the semicolon identifies the statement to be executed next if the relation has the value true. If the relation is false, we assume the next statement in sequence will be the one executed next.

Other statements: A large variety of other kinds of statements are found in actual programming languages. Our task in this chapter will be large enough (and realistic enough) if we limit ourselves to the following:

START	corresponds to the "start box", performs no operation but can have a label which would be the name of the program.
STOP	corresponds to the "stop box", indicates the end of calculation and either stops the computer or returns control to a monitor program.
READ	corresponds to the "input box", to be followed by the input list.
PRINT	corresponds to the "output box", to be followed by the output list.
GO TO	does not correspond to any flow chart box. It directs an unconditional change in the execution sequence to the statement the label of which follows the words "GO TO".
END	does not correspond to any flow chart box. It marks the physical end of the deck of cards containing the program.

[†] We do not prohibit the key word from being used in another context as an identifier. Many actual programming languages do, in fact, make such a restriction in the interest of a more efficient compiling operation or simply to avoid confusion when the same string of symbols can mean different things in a source program.

Example

We give in Figure 8-8 a TYPICAL program corresponding to the process given in Figure 3-7 for tallying low, mid and high grades.

```
1              START; READ N;
2              COUNT ← 1; LOW ← 0; MID ← 0; HIGH ← 0;
3       LOOP:  READ T;
4              IF (T > 50) THEN TEST2; LOW ← LOW + 1; GO TO INC;
5     TEST 2:  IF (T > 80) THEN GO ON; MID ← MID + 1; GO TO INC;
6      GO ON:  HIGH ← HIGH + 1;
7        INC:  COUNT ← COUNT + 1; IF(COUNT ≤ N) THEN LOOP;
8              PRINT LOW, MID, HIGH; STOP; END;
```

Figure 8-8. A TYPICAL program

The lines in this program are numbered only for our reference. Line numbers are not a part of the program. Capital letters have been used throughout. (The alphabets of most procedural languages are unfortunately limited to capital letters.)

Notice the frequent appearance of semicolons; a semicolon at the end of every statement, in fact. Line 1 has the start statement and a read statement assigning to N the number of grades to be tallied. Use of the semicolon allows us to put several statements that may be associated with each other on the same line. Of course, the programmer could put each statement on a separate line. It is mainly a matter of taste.

Line 3 illustrates a statement label, LOOP, followed by a colon. This statement reads a card containing a value which is assigned to the variable T.

For each test value T lines 4 and 5 determine whether that grade is in the LOW, MID or HIGH group. The relations used on lines 4 and 5 are the reverse of those used in the flow chart. This reversal is suggested by the form of the TYPICAL conditional statement. The rest of the program shown should explain itself.

Two of the statements in TYPICAL do not correspond to any box of our flow chart language and, indeed, are not necessary. They are included simply because, in practice, many languages do have such statements. Strictly speaking, a GO TO is unnecessary since a statement of the form

IF (A = A) THEN label;

has the same effect because $A = A$ is always true. An END statement is not necessary because its only purpose is to mark the end of the deck of cards containing the program. The same purpose could be achieved by asking, "Are there more data?" since these statements are data to the compiler program.

Frequently computers (especially larger ones) use a "monitor" which is a control program over the operation of the computing system. Monitor programs will sequence the various jobs (probably each job is prepared by a different person) to be performed. They will identify whether the job is a machine language execution or requires compilation (and if so, in what language). They will select the necessary compiler program and library programs. They will time the length of the job run and produce whatever statistical and accounting reports are required. When operated with a monitor program, the computer rarely stops. STOP in such instances doesn't mean "stop the computer". Instead, it is an order to release control of the computer to the monitor program. The monitor will then be able to accept the next job which is waiting to be processed. The function of the END statement can also be assimilated in the monitor program. Nevertheless, most languages require some end-of-program indication as part of the program.

8-4 Prescan (the preliminary steps of a compiler)

As we shall conceive of it, the prescan portion of the compiler identifies statements as to type and arranges statements into a standard form to facilitate further processing, such as decomposition of assignments. A major function of prescan, which we gloss over here, is to analyze each statement to determine if it conforms to the syntax (i.e., grammar) of the source language. Actual compilers frequently contain prescan diagnostic programs which can produce numerous kinds of messages (called "diagnostics") to name syntax violations. In the compiler we are discussing, it is assumed that all incoming statements have been properly written, admittedly an unrealistic assumption.

Consider, now, what happens when a program written in TYPICAL is to be translated. First, the cards containing TYPICAL statements are read, one at a time, into the computer. Each card contains a string of characters. When a card is read, its content is assigned to a string $\overline{S}$ in memory. When the next card is read by the same instructions, its content is also assigned to $\overline{S}$, destroying what was read from the previous card. Therefore, between the reading of successive cards into the computer we should scan $\overline{S}$, saving whatever parts are needed for further processing.

The natural subdivision of a program to use in processing is the statement. This means that the string of characters must be scanned for the semicolon. If a semicolon is not found, the statement must continue on following cards. If a semicolon is found, the statement ends on the current card although it may have begun on an earlier card. Figure 8-9 illustrates a few of the ways a complete statement might be assembled from one or more cards.

When the semicolon marking the end of a statement is found, its position must be remembered since the next statement could begin in the next position of the same card, as, for example, on line 1 of Figure 8-8. During the process of assembling the string $\overline{T}$ from the one or more $\overline{S}$'s, we find it relatively convenient to inspect the characters being moved for the purpose of deleting blank spaces.

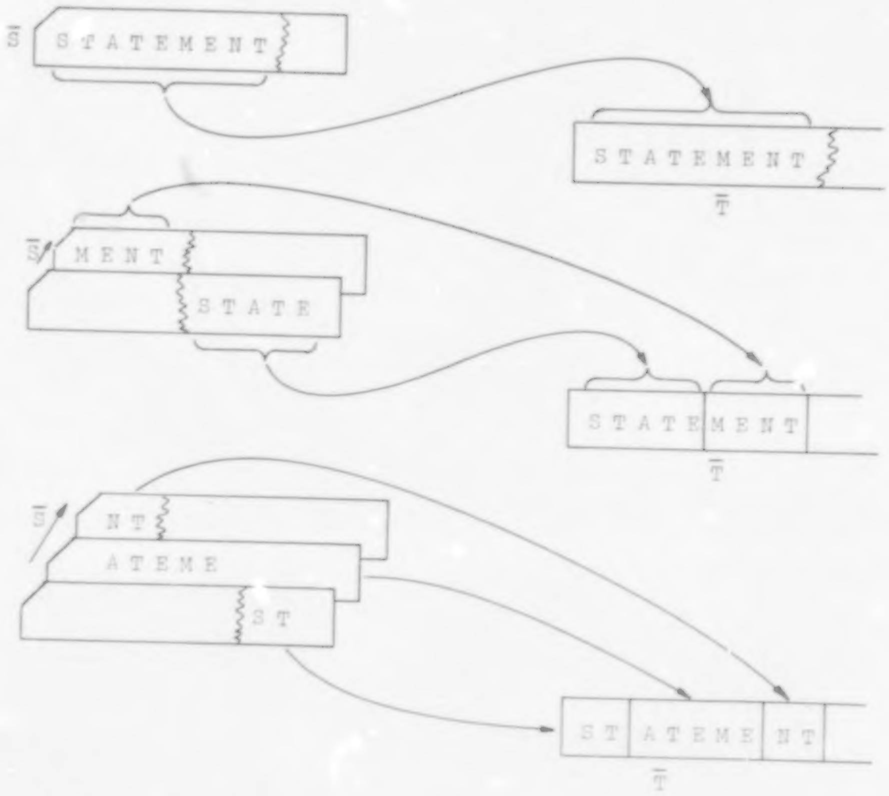

S̄
STATEMENT
STATEMENT
T̄
S̄
MENT
STATE
STATEMENT
T̄
S̄
NT
ATEME
ST
STATEMENT
T̄

Figure 8-9. Assembly of parts of a statement which have been read from one or more cards

Example 1 (Assembly of statements)

We will prepare a flow chart for a program to read successive cards containing TYPICAL statements, to move (and assemble, if necessary) a single statement for "safekeeping", and to eliminate blank spaces during the move process.

A general flow chart like the one in Figure 8-10 is frequently the first stage in the organization of a problem. The object is to show the overall flow of information without concern for detail. We will study this flow chart critically to see what it does do and what it fails to do.

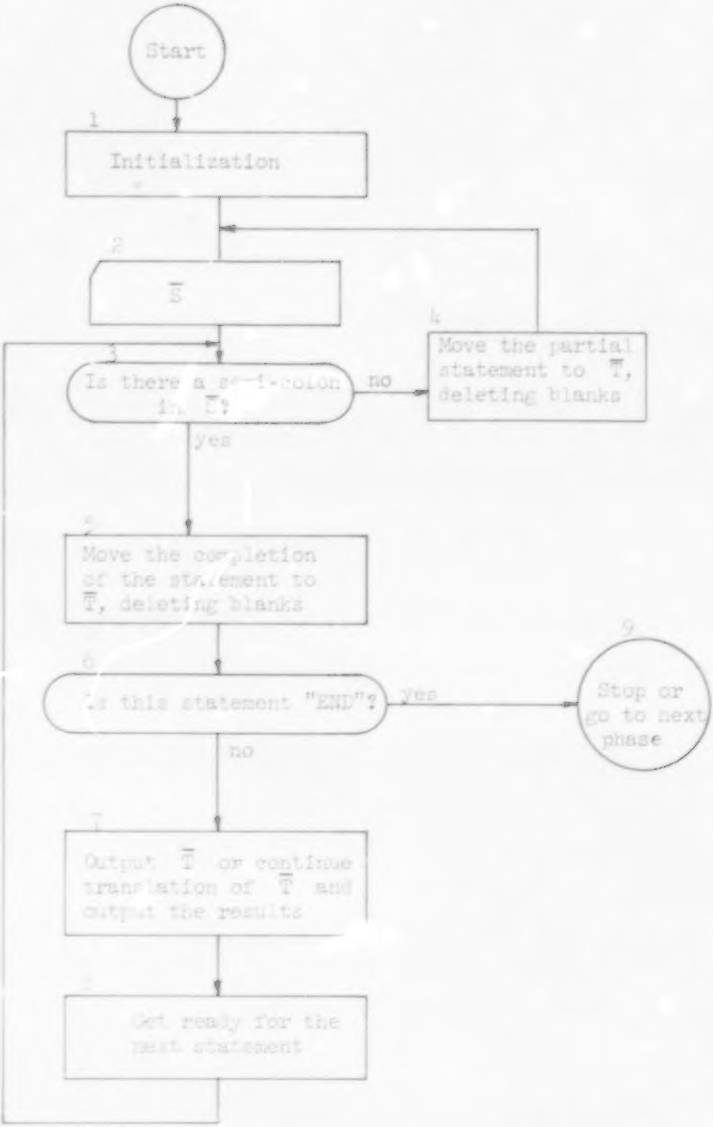

Start
1
Initialization
2
$\overline{S}$
3
Is there a semi-colon in $\overline{S}$?
no
4
Move the partial statement to $\overline{T}$, deleting blanks
yes
5
Move the completion of the statement to $\overline{T}$, deleting blanks
6
Is this statement "END"?
yes
9
Stop or go to next phase
no
7
Output $\overline{T}$ or continue translation of $\overline{T}$ and output the results
8
Get ready for the next statement

Figure 8-10. A general flow chart for assembling TYPICAL statements

Box 1 is intended to provide for assignment of initial values to counters, pointers and whatever else needs setting at the beginning of the process. The details of initialization are filled in as the details of other boxes in the flow chart become known. (Many people habitually include such a box in a general flow chart as a reminder that initial values invariably will need to be assigned.)

Box 2 reads a card, assigning the content of the card to string $\overline{S}$. Each time a card is read the previous content of $\overline{S}$ is destroyed.

Box 3 asks whether a semi-colon, indicating the end of a statement, appears in $\overline{S}$. If there is no semi-colon in $\overline{S}$, the statement currently being assembled must continue on the next card to be read. In this case, Box 4 moves the part of the current statement which is in $\overline{S}$ to $\overline{T}$ with the deletion of blank spaces and the flow chart returns to read another card. Box 4 is the type of job for which the delete procedure of Section 8-3 was designed.

On the other hand, if Box 3 does find a semi-colon in $\overline{S}$, we know that the statement currently being assembled has ended. Whatever part of the current statement is in $\overline{S}$ (whether it is a complete statement or only the "tail-end") should be moved to $\overline{T}$ with the deletion of blank spaces. This is done by Box 5.

When Box 6 of Figure 8-10 is reached we know that an entire statement has been assembled (with spaces deleted) in $\overline{T}$. Box 6 determines whether the statement assembled in $\overline{T}$ is the END statement so that a transfer to Box 9 can be made to stop (or go on to the next phase of the program).

If the statement in $\overline{T}$ is not the END statement, we would want to go through the same process for the next statement. However, we must first do something to preserve the statement already in $\overline{T}$ (lest it be destroyed by the next statement). Box 7 indicates that $\overline{T}$ might be output or the compiler might be organized so that further processing could be done before $\overline{T}$ is output.† That is, Box 7 could stand for a major section of program.

Box 8 is our reminder (like Box 1) to be sure to do any incrementing or to set any pointers we may need before returning to Box 3 to examine what is left of $\overline{S}$.

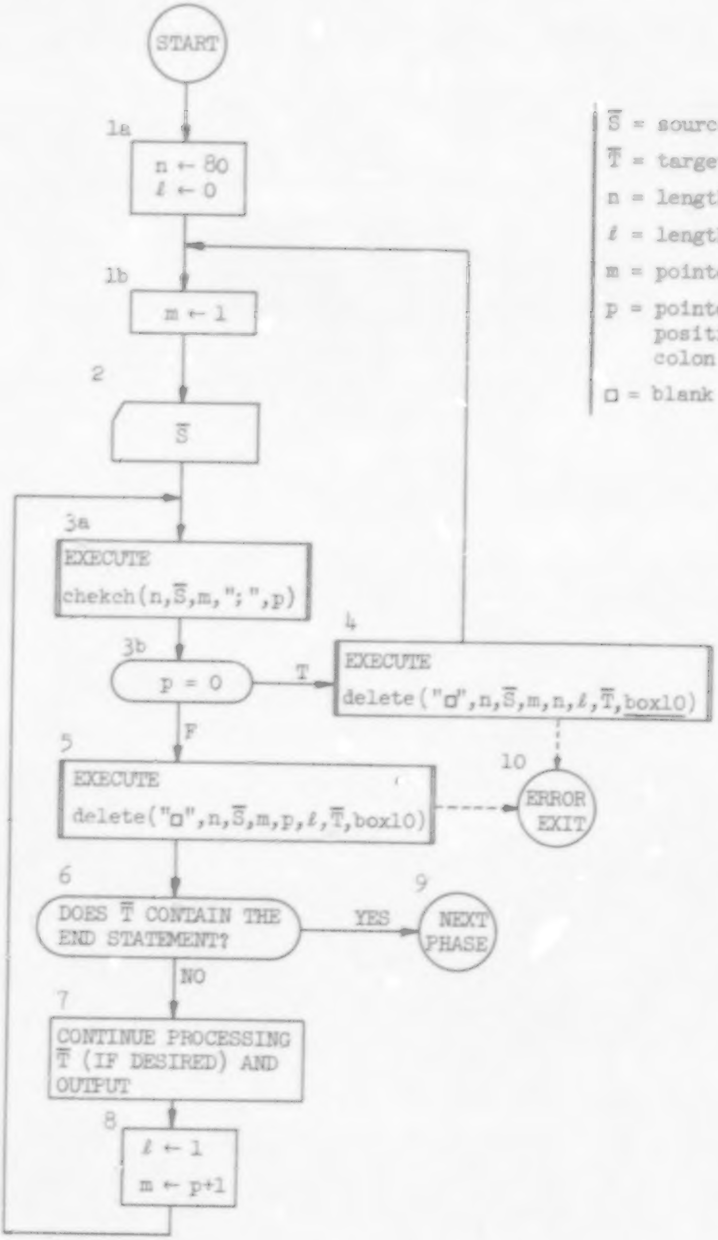
Figure 8-11 shows how the major parts of this flow chart can be detailed.

---

†Output should be in a machine readable form such as punched cards or magnetic tape.

During a first reading you may wish to skip over this detail and proceed to a discussion of the next phase which begins at Example 2.

Box 1 of Figure 8-11 has been divided into two parts so that each time we return to read another card we can set a pointer m to the first character of $\overline{S}$. A card is read filling $\overline{S}$ with 80 characters in box 2. The chekch procedure is used (box 3a) to search for a semicolon in $\overline{S}$ and the delete procedure is used to develop the target string $\overline{T}$, whether or not the semicolon is found. Boxes 6 and 7 are not detailed.


START
1a
n ← 80
ℓ ← 0
1b
m ← 1
2
S̄
3a
EXECUTE
chekch(n,S̄,m,";",p)
3b
p = 0
T
4
EXECUTE
delete("□",n,S̄,m,n,ℓ,T̄,box10)
F
5
EXECUTE
delete("□",n,S̄,m,p,ℓ,T̄,box10)
10
ERROR EXIT
6
DOES T̄ CONTAIN THE END STATEMENT?
YES
9
NEXT PHASE
NO
7
CONTINUE PROCESSING T̄ (IF DESIRED) AND OUTPUT
8
ℓ ← 1
m ← p+1

$\overline{S}$ = source string

$\overline{T}$ = target string

n = length of string $\overline{S}$

ℓ = length of string $\overline{T}$

m = pointer scanning $\overline{S}$

p = pointer marking the position of a semicolon

□ = blank space

Figure 8-11. Partially detailed flow chart for assembling TYPICAL statements

379

We leave this example unfinished, but we hope the interested student will wish to complete it.

Example 2 (Identifying statement types)

A compiler must be able to tell what kind of statement it is inspecting so that the appropriate procedures can be used in its analysis and translation. In this example we will show how the identification of statement type could take place.

In box 6 of Figure 8-10 we have already raised the question of identifying an END statement. In addition, we now want to identify assignment statements and others such as STOP, READ, etc. Box 6 of Figure 8-10 could be replaced by Figure 8-12.

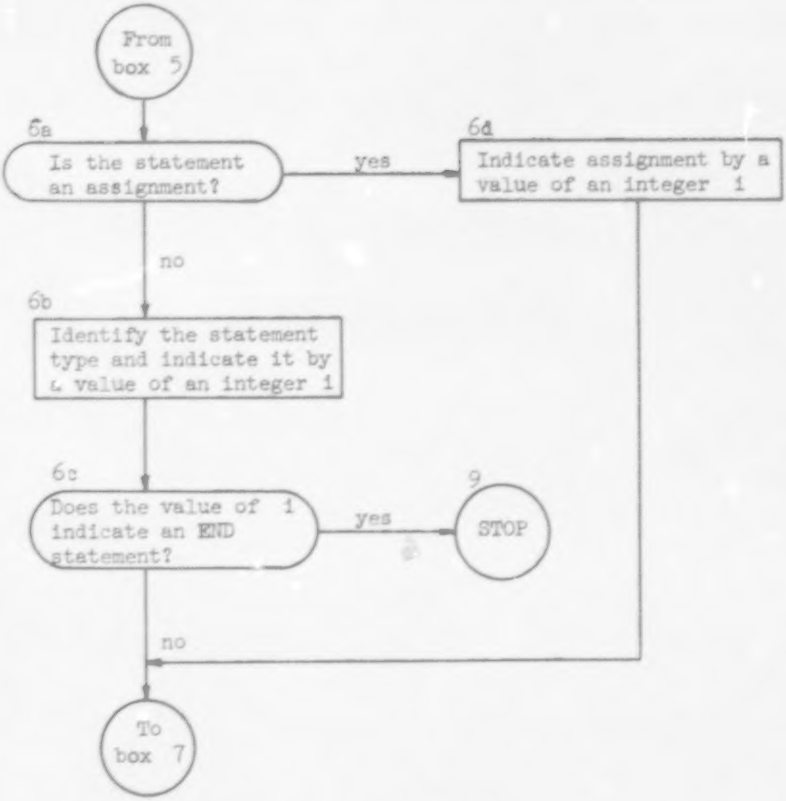

From box 5
6a
Is the statement an assignment?
yes
6d
Indicate assignment by a value of an integer i
no
6b
Identify the statement type and indicate it by a value of an integer i
6c
Does the value of i indicate an END statement?
yes
9
STOP
no
To box 7

Figure 8-12. An amendment to Figure 8-10 to identify the type of each statement

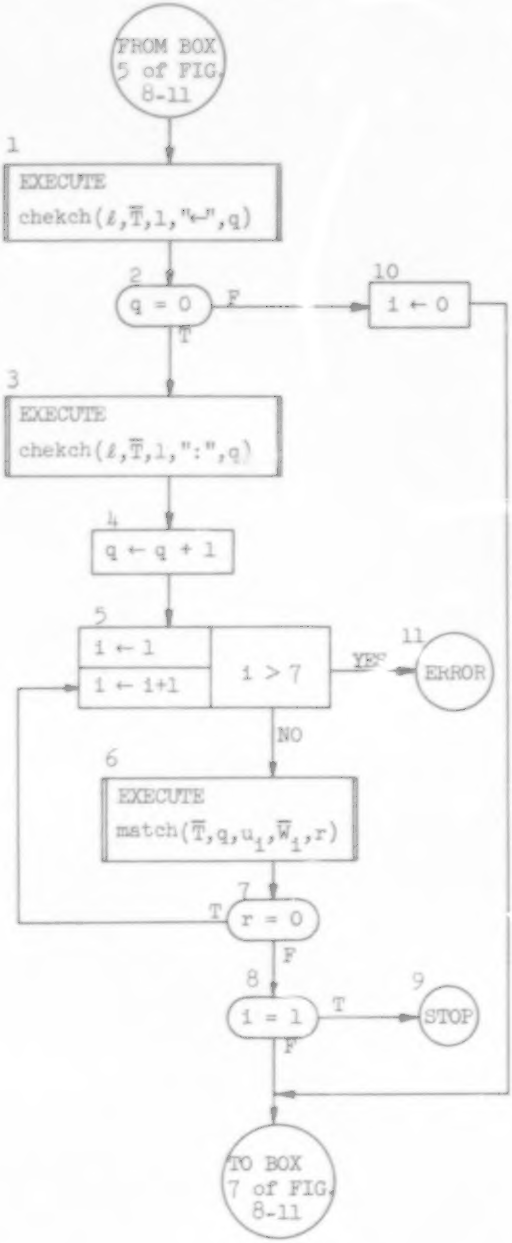
All statements, except assignment statements, are characterized by descriptive or key words. Therefore, it is natural to identify the assignment statement (by looking for a left-pointing arrow) separately in box 6a. The result of the identification will be to give some value to an integer i. For example, if the statement is an assignment statement, we can set i to zero in box 6d.

Box 6b has to locate the key word for each statement. This is easy since it starts either at the beginning of the string or immediately after the colon separating a label from the rest of the statement. Once located, the key word is to be successively compared with each of a list of permitted key words. When an equal comparison is found, i can be set equal to the index of the list of permitted key words. The details for these steps, given in Figure 8-13 and the next paragraph of text, may be skipped on your first reading.

At this point you have seen enough flow charts similar to Figure 8-13 that little explanation should be needed, especially since it corresponds closely to Figure 8-12 and that has been discussed in the last two paragraphs. The one really new thing in this flow chart is the reference (in box 6) to a list of strings (key words) of different lengths. In the initialization, or built into the program as constants, there must be a definition of the u_i and W_i $(1 \leq i \leq 7)$. Figure 8-13 assumes that we have defined

$$W_1 \leftarrow \text{"END"} \quad \text{and} \quad u_1 \leftarrow 3$$

so that the test in box 8 is the actual test for an END statement.


FROM BOX 5 of FIG. 8-11
1
EXECUTE
chekch(ℓ,$\overline{T}$,1,"←",q)
2
q = 0
F
T
10
i ← 0
3
EXECUTE
chekch(ℓ,$\overline{T}$,1,":",q)
4
q ← q + 1
5
i ← 1
i ← i+1
i > 7
YES
11
ERROR
NO
6
EXECUTE
match($\overline{T}$,q,u_i,$\overline{W}_i$,r)
7
T
r = 0
F
8
i = 1
T
9
STOP
F
TO BOX 7 of FIG. 8-11

ℓ = length of string $\overline{T}$

$\overline{T}$ = string containing statement

i = integer set to indicate type of statement

q = pointer

r = indicator of success of the match

$\overline{W}_i$ = i^{th} descriptive word

u_i = length of $\overline{W}_i$

Figure 8-13. A detailed flow chart for statement type identification

When the computer is asked to scan and analyze each statement (especially assignment statements), it will be most convenient if we can have the statement look to the computer as a string of equal length elements, each occupying, for example, one word in memory. Each element should be a unique item or maybe (in the simplest sense) even a single unique character. The way it is now, the elements of a statement can vary in size (number of characters). Thus, key words have different lengths and so do identifiers, like labels and variables, and so do constants.

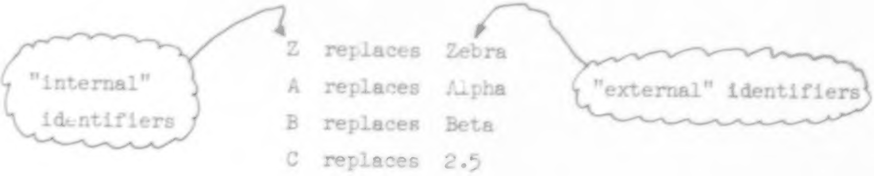
A straightforward way to make this change is to replace each item in a statement by some unique character. For example,

$$\text{Zebra} \leftarrow \text{Alpha} + (2.5/\text{Beta});$$

could be replaced by

$$Z \leftarrow A + (C/B);$$

where



Z replaces Zebra
A replaces Alpha
B replaces Beta
C replaces 2.5

Clearly we won't be able to replace many identifiers if we are limited to the 26 letters of the alphabet. Additional codes for string elements are needed and can be defined but we will not worry about them because the codes often depend on the characteristics of the computer used. As one simple example of such a code we could just bracket the identifier with a character not permitted in TYPICAL source programs and so clearly distinguishable (e.g., the dollar sign $).

In addition to the desirability of being able to refer to an identifier as a single element of a string of elements, other characteristics of the identifier can profitably be coded in the new symbol. Examples of such characteristics are whether the identifier is a real constant, an integer constant, a real variable, an integer variable, etc. For the purpose of this explanation such coding questions are an unnecessary complication (although useful in practice).

Example 3 (Replacing an assignment statement by a string of equal-length elements)

Prepar[illegible] chart for a process which substitutes a unique internal identifier for each distinct external identifier of an assignment statement.

Solution of this example problem is best reached by the same kind of process we have used before. We first sketch general flow charts, then refine the various boxes until we have a detailed flow chart.

We will want to produce a table showing the correspondence between internal and external identifiers. Note that the external identifier may already be in the table when it is encountered in the statement so the table of correspondence has to be examined before we can assign a new internal identifier.

In scanning a statement from left to right, only certain symbols can follow an identifier. We list these and call them "isolators":

$$\leftarrow \quad + \quad - \quad / \quad \times \quad \uparrow \quad) \quad : \quad ;$$

Notice the left parenthesis is not in this list. The problem is to search for these symbols which isolate the intervening identifiers or constants. When found, the identifier or constant will be replaced by an internal identifier. What happens when we encounter a left parenthesis will be explained later.

Let:

$\overline{T}$ be the assignment statement represented as a string of length n.

$\overline{G}$ be the generated string of internal identifiers.

Ext be the list of external identifiers, each identifier being a different string of characters and not all the same length.

Ext_k be the k^{th} string in a list of external identifiers having a length d_k.

Int_k be the k^{th} entry in a list (Int) of internal identifiers.

$\overline{\text{DEL}}$ be a string of length 9, containing the nine isolators.

i be a scanning pointer on $\overline{T}$.

j be a place-holding pointer on $\overline{T}$.

k be a counter indicating the current length of Ext (or Int).

ℓ be a pointer indicating the next position of $\overline{G}$.

A diagram of information flow, Figure 8-14, may help in illuminating the problem. The original string $\overline{T}$ is to be scanned until the first isolator (in TYPICAL either : or ←) is encountered. The first external identifier is placed in the list of external identifiers (Ext) if it is not already in this list, and the corresponding internal identifier (from Int) becomes the first element of the generated string $\overline{G}$. The first isolator, "←" in this case, is then transferred directly from $\overline{T}$ to $\overline{G}$.

The scan of $\overline{T}$ continues until the next isolator is found. In this case it is "×". Remember, a left parenthesis is not an isolator. One must then ask whether or not the character immediately following the previous indicator is a left parenthesis. If so, it is transferred to $\overline{G}$ and a pointer moved to the next element of $\overline{T}$. The same question (left parenthesis?) should be asked repeatedly in case there are nested parentheses. When that character is not a left parenthesis, it must be the first character of an external identifier. The identifier is then entered into Ext (after checking that it is not already in the Ext list).

With this picture in mind we can better understand the general flow of Figure 8-15. The transition from the general flow chart to a detailed flow chart is a process of deciding how to implement each statement and each question in detail. In a problem of this size, it can be helpful to use an intermediate stage in which the easier implementations are specified but the more complicated implementations are left as general questions or statements. An intermediate flow chart is shown next in Figure 8-16.

After studying the general flow chart in Figure 8-15 you might find it more profitable to skip the details (the rest of Section 8-4) and return to it during a second reading.

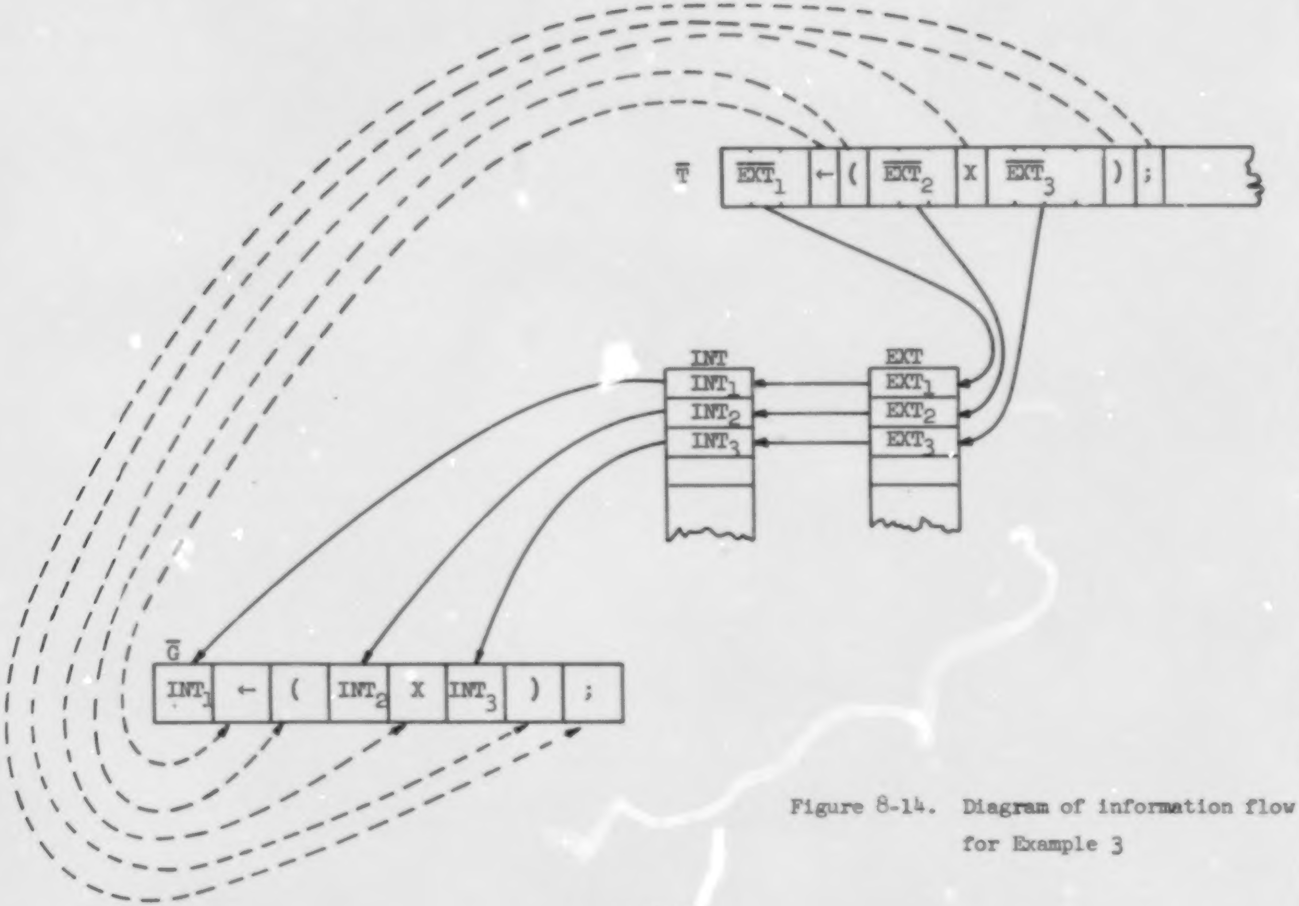

T̄
EXT̄1
←
(
EXT̄2
X
EXT̄3
)
;
INT
INT1
INT2
INT3
EXT
EXT1
EXT2
EXT3
Ḡ
INT1
←
(
INT2
X
INT3
)
;

Figure 8-14. Diagram of information flow for Example 3

j = pointer indicating first possible element of an identifier in $\overline{T}$.

k = length of Ext (or Int) list

i = pointer scanning $\overline{T}$ for isolators.

ℓ = pointer indicating the next position of $\overline{G}$

$\overline{T}$ = assignment statement (source string)

$\overline{G}$ = generated string

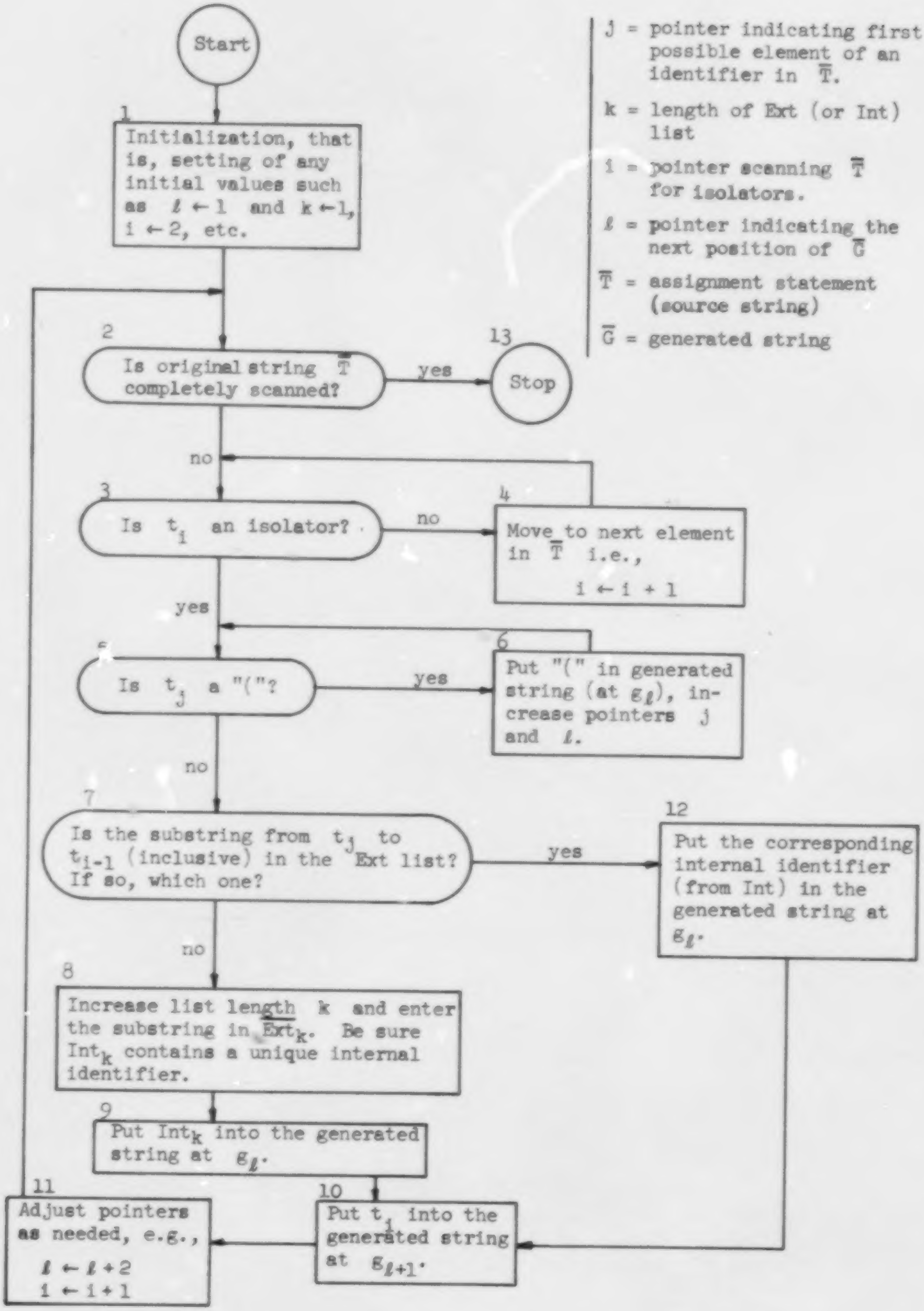
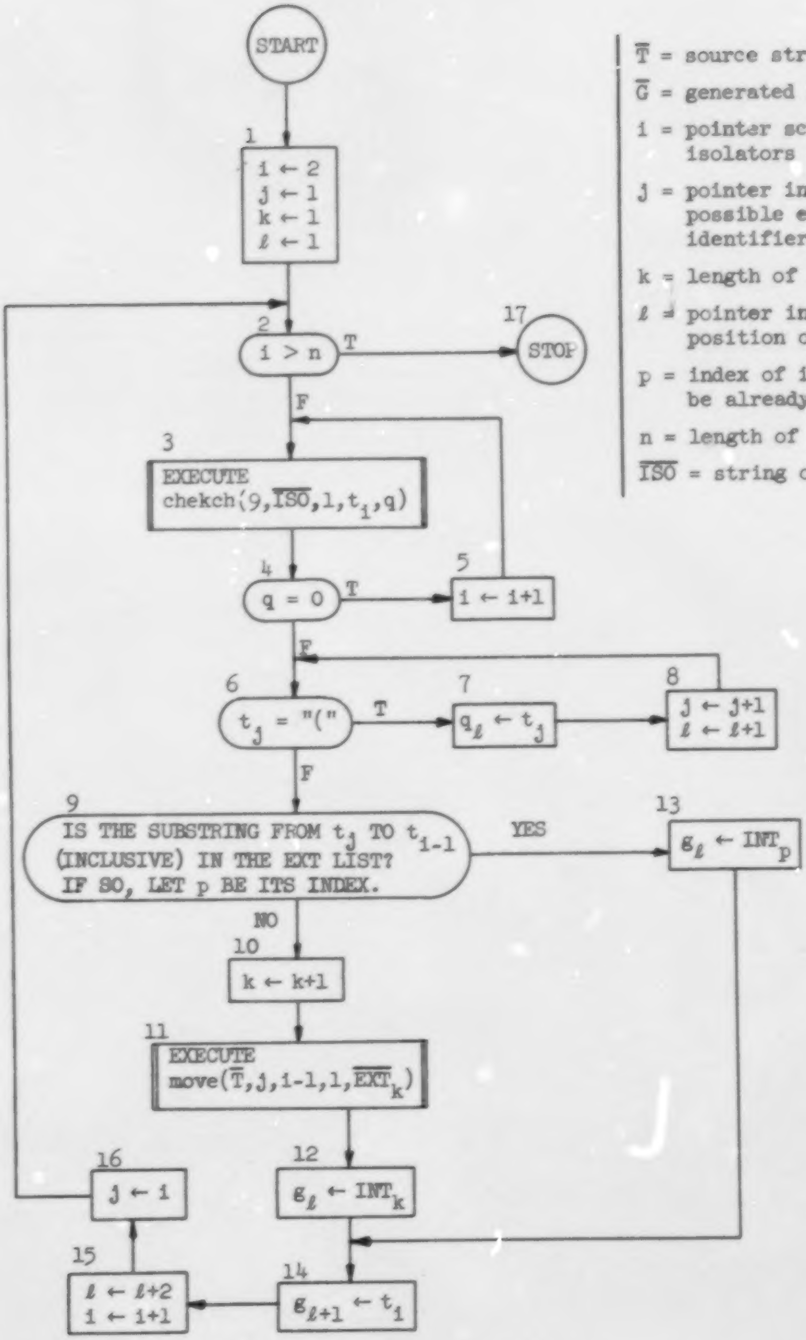

Start
1
Initialization, that is, setting of any initial values such as ℓ ← 1 and k ← 1, i ← 2, etc.
2
Is original string T̄ completely scanned?
yes
13
Stop
no
3
Is tᵢ an isolator?
no
4
Move to next element in T̄ i.e., i ← i + 1
yes
5
Is tⱼ a "("?
yes
6
Put "(" in generated string (at gℓ), increase pointers j and ℓ.
no
7
Is the substring from tⱼ to tᵢ₋₁ (inclusive) in the Ext list? If so, which one?
yes
12
Put the corresponding internal identifier (from Int) in the generated string at gℓ.
no
8
Increase list length k and enter the substring in Extₖ. Be sure Intₖ contains a unique internal identifier.
9
Put Intₖ into the generated string at gℓ.
10
Put tᵢ into the generated string at gℓ₊₁.
11
Adjust pointers as needed, e.g., ℓ ← ℓ + 2 i ← i + 1

Figure 8-15. General flow chart for Example 3


START
1
$i \leftarrow 2$
$j \leftarrow 1$
$k \leftarrow 1$
$\ell \leftarrow 1$
2
$i > n$
T
17
STOP
F
3
EXECUTE
chekch(9,$\overline{ISO}$,1,t_i,q)
4
q = 0
T
5
$i \leftarrow i+1$
F
6
t_j = "("
T
7
$q_\ell \leftarrow t_j$
8
$j \leftarrow j+1$
$\ell \leftarrow \ell+1$
F
9
IS THE SUBSTRING FROM t_j TO t_{i-1} (INCLUSIVE) IN THE EXT LIST? IF SO, LET p BE ITS INDEX.
YES
13
$g_\ell \leftarrow INT_p$
NO
10
$k \leftarrow k+1$
11
EXECUTE
move($\overline{T}$,j,i-1,1,$\overline{EXT}_k$)
12
$g_\ell \leftarrow INT_k$
16
$j \leftarrow i$
15
$\ell \leftarrow \ell+2$
$i \leftarrow i+1$
14
$g_{\ell+1} \leftarrow t_i$

$\overline{T}$ = source string

$\overline{G}$ = generated string

i = pointer scanning $\overline{T}$ for isolators

j = pointer indicating first possible element of an identifier in $\overline{T}$

k = length of Ext (or Int) list

ℓ = pointer indicating the next position of $\overline{G}$

p = index of identifier found to be already present in Ext

n = length of string $\overline{T}$

$\overline{ISO}$ = string of isolators

Figure 8-16. Intermediate flow chart for Example 3.

The way that the chekch procedure is used in Box 3 of Figure 8-16 is of some interest. This use of the chekch procedure is different from other uses of it we have seen. In this case we are trying to find out whether an element of unknown value is one of a set of known values. Previously we have used chekch to find out if a known element was in a set of unknown values.

Actually the only undetailed box of Figure 8-16 is the question of whether the substring is already contained in the Ext list of strings (in Box 9). You will find the match procedure helpful at this point. When we do try to give a detailed chart for this box we discover a need (extremely common when drawing a complicated flow chart) to know the length of each $\overline{\mathrm{Ext}}_k$ string. Even though these lengths, d_k, have not been provided for in Figure 8-16, they are easily accommodated by the following alteration to replace Boxes 9 and 10.

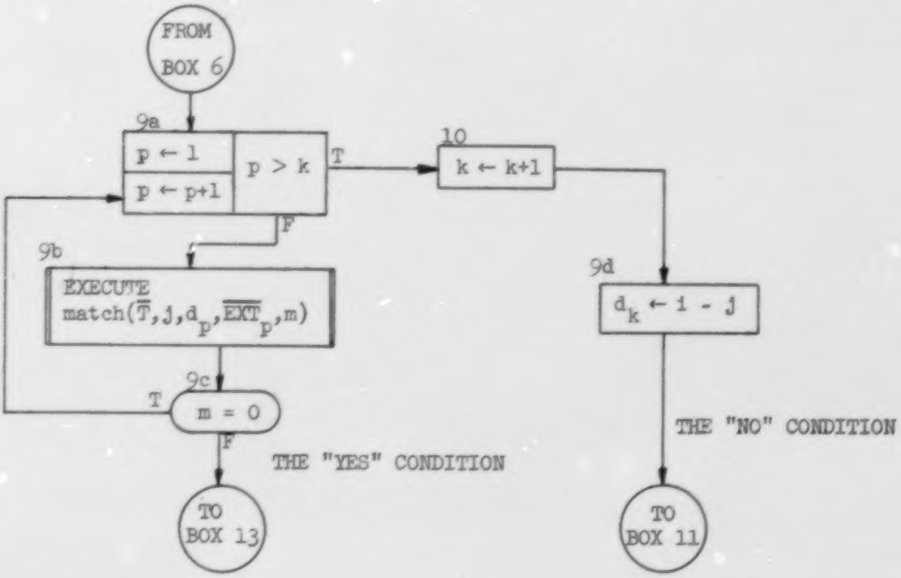

FROM BOX 6
9a
p ← 1
p ← p+1
p > k
T
F
10
k ← k+1
9b
EXECUTE match($\overline{T}$,j,d_p,$\overline{EXT}_p$,m)
9c
m = 0
T
F
THE "YES" CONDITION
TO BOX 13
9d
d_k ← i - j
THE "NO" CONDITION
TO BOX 11

Figure 8-17. Detailed insertion to complete Figure 8-16

Insertion of Figure 8-17 into Figure 8-16 gives a detailed flow chart solution.

8-5 The Decomposition of Assignment Statements

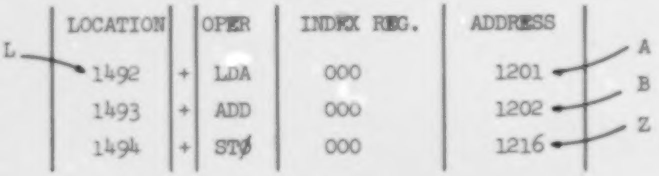
A compiler develops a sequence of instructions in machine code for each statement of the source language program. In some cases, such as the trivial case of a STOP statement being transformed to a "HLT" instruction, it is easy to see how this translation is done. It is even easy to see how rules can be stated so that a simple assignment statement can be automatically decomposed. For example, a compiler which translates from TYPICAL to SAMOS could take the statement:

$$L : Z \leftarrow A + B;$$

and decompose it into

LLLL	LDA	000	AAAA
	ADD	000	BBBB
	STO	000	ZZZZ

where LLLL, AAAA, BBBB and ZZZZ are internal identifiers for L, A, B and Z respectively. One can also see that LLLL, AAAA, BBBB and ZZZZ can be associated with definite addresses in memory in some organized way so that the final translation might be, for example:



LOCATION		OPER	INDEX REG.	ADDRESS
1492	+	LDA	000	1201
1493	+	ADD	000	1202
1494	+	STO	000	1216

Figure 8-18. SAMOS Instructions for $L : Z \leftarrow A + B;$

It is not so easy to see how one can find an algorithm to decompose any large, messy (but properly written) expression such as

$$Z \leftarrow ((A \times B + C)/C \uparrow 2 - (B - A))/(A \uparrow 2 + B \uparrow 2) \uparrow 2$$

into a series of machine steps. Where would one start? One approach to this question is suggested in Section 2-4 and amplified in Appendix B.

Basically the problem is to decide on the order in which operations are to take place. The problem of automatic decomposition would be solved (or nearly so) if we could discover an unambiguous way to determine the ordering of operations.

Exercises 8-5 Set A

List the order in which you would do the operations for each of the following. Identify the reasons for your choice of orderings.

(a) $A + B \times C$
(b) $A \times B + C$
(c) $(A + B) \times C$
(d) $A \times (B + C)$
(e) $A + B \uparrow C \times D$
(f) $A + B \uparrow (C \times D)$
(g) $(A + B) \uparrow C \times D$
(h) $A \uparrow B + C \times D$
(i) $A \uparrow (B + C) \times D$
(j) $A \uparrow (B + C \times D)$

Two potential methodical approaches for ordering the operations appear possible. (1) We could try to find a way to add a sufficient number of parenthesis pairs, i.e., to fully parenthesize an expression, so that precedence rules for operators would not be important. (2) We could use the precedence rules to eliminate parentheses entirely. Both approaches have been studied. The second (that of eliminating parentheses) has turned out to be superior since it is not only more efficient in operation but it is also aesthetically more pleasing.

It has long been known that expressions could be written without the use of parentheses[†]. The technique that we will use is called postfix notation since a binary operator is written after (post) its operands. For example,

$a + b$ is written $ab+$
$a \times b$ is written $ab\times$, and so on.

[†] A Polish mathematician, Lukasiewicz, first defined such notation. Consequently, such notations are commonly called "Polish notation" or "Lukasiewicz notation".

To illustrate, consider Exercises 8-5, (a) through (d). In postfix notation,

$A + B \times C$ is written as $ABC\times+$

$A \times B + C$ is written as $AB\times C+$

$(A + B) \times C$ is written as $AB+C\times$

$A \times (B + C)$ is written as $ABC+\times$

Exercise 8-5 Set B

How would the expressions of Exercise 8-5, Set A, (e) through (j) be written in postfix notation?

When an expression is written in postfix notation the ordering of operations is uniquely determined by their left-to-right order of occurrence. For example, we claim that the (more pleasing?) expression,

$$ZAB \times C + C2 \uparrow /BA--A2 \uparrow B2 \uparrow + 2 \uparrow/\leftarrow$$

is the postfix form of

$$Z \leftarrow ((A \times B + C)/C \uparrow 2 - (B - A))/(A \uparrow 2 + B \uparrow 2) \uparrow 2$$

The postfix form may look even worse to you because you are not used to seeing expressions written this way. Nevertheless, its meaning is unambiguous. It tells us, reading it left to right, to:

		Accounts for
1.	multiply A by B	$AB\times$
2.	to this add C	$AB\times C+$
3.	save that result (call it α) and square C	$C2\uparrow$
4.	divide α by the square of C	$AB\times C+C2\uparrow/$
5.	save the result as α and subtract A from B	$BA-$

etc. (The student should complete this list of instructions.)

Formalization of the rules for the interpretation of any expression written in postfix notation is not difficult.

As a list of rules:

1. Scan the string from left to right to find the first operator,
2. Execute that operation using the two immediately preceding operands,
3. Replace the three symbols used by the result of the operation (a new operand),
4. Return to Step 1 until no operators remain.

We see this list of rules put in flow chart form in Figure 8-19.

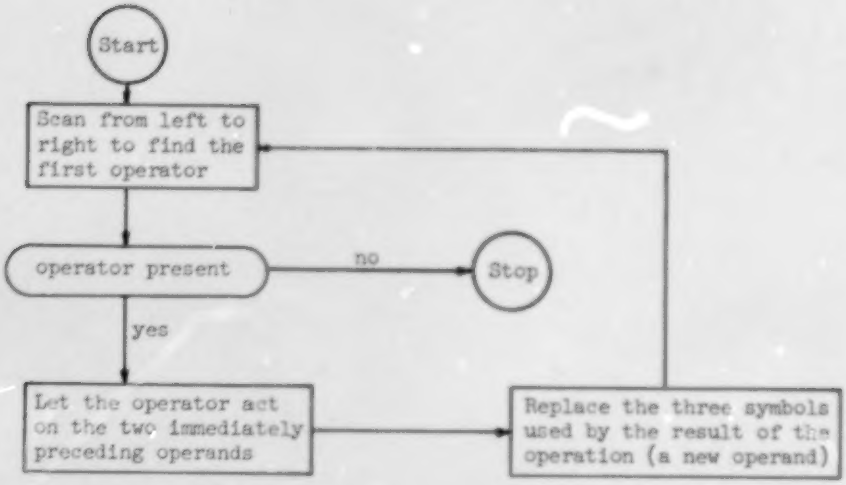

Start
Scan from left to right to find the first operator
operator present
no
Stop
yes
Let the operator act on the two immediately preceding operands
Replace the three symbols used by the result of the operation (a new operand)

Figure 8-19. General rule for interpretation of postfix form

Our job is thus reduced to transforming the flow chart of Figure 8-19 into an algorithm sufficiently detailed that it can be written in flow chart language. (From there to a programming language would be only a small step.)

First we introduce an expanded precedence scale or table that covers all the "isolators" that can appear in an assignment statement (Figure 8-20). Consulting this table properly assists us in eliminating parentheses.

			Precedence "value"
	↑		6
	× /		5
	+ -		4
	)	Increasing order of precedence	3
to be explained later	←		2
	:		1
	;		0

Figure 8-20. Precedence table

In general, this table can be interpreted as meaning that operations having a higher precedence value should be performed before those having a lower precedence value. But we know that it is not as easy as that. In the past we have thought of parentheses as intervening to limit the length of the string over which the simple precedence rule can be applied. The secret to overcoming this barrier is to think of the right parenthesis as an operator, one which has a precedence value that is lower than that of any arithmetic operator. We keep in mind that whenever a pair of parentheses enclose a single identifier, like "(A)", such redundant parentheses will somehow be recognized and removed.

Now consider neighboring pairs of operators as, for example, in the expression

(A ↑ 2 + B ↑ 2) ↑ 2 .

The neighboring pairs of operators are [↑,+], [+,↑], [↑,)] and [),↑]. The left parenthesis is not considered an operator, so it is, of course, not represented in any operator pair. Notice that, for the first and third pair, the left operator has a higher precedence than the right operator. These left operators are the ones we would naturally do first. Do we have a clue to a rule for ordering operations?

Consider a second example:

A × (B + C) .

Now the neighboring pairs of operators are [×,+] and [+,)]. The left operator of each pair has a higher precedence than the right operator but we know that we should do the addition first. What tells us that the multiplication is not to be done first? Note that a left parenthesis occurs between the multiplication and the addition. The effect of a left parenthesis *between* a pair of operators should be to eliminate that pair of operators as a choice.

We will now state a tentative rule and see how it works by trying it out on (A ↑ 2 + B ↑ 2) ↑ 2.

Tentative rule

1. Make a list of all neighboring operators *which are not separated by a left parenthesis*.
2. Choose the first (leftmost) pair of operators for which the left operator has a higher precedence than the right operator.
3. Select the left operator of the chosen pair for execution.

4. Replace the "triple" consisting of the chosen operator and its left and right operands by a special "placeholder".

5. Take the triple removed in step 4 and rearrange it into postfix form.

6. Generate the postfix string by appending the postfix form produced in step 5 to whatever string has already been generated.

7. Repeat steps 2 through 6 until no more operator pairs remain.

Application of the tentative rule to $(A^2 + B^2)^2$ is displayed, step-by-step, in Figure 8-21. In the first line, it happens that the first pair of operators is selected so that we are concerned with the first exponentiation. The triple for this operator is $A \uparrow 2$ which is now replaced by the placeholder, α, as shown in Line 2, Column 1. The postfix form is $A2\uparrow$. This is added to the generated string which was initially empty.

The second line of Figure 8-21 proceeds in a very similar way. Here the second pair of operators is selected. The third line goes in much the same way (the first pair of operators being selected) until we attempt to append the postfix form, $\alpha\beta+$, to the generated string. Then we should ask ourselves if a placeholder can appear in the generated string (the output of the procedure).

α is a symbol standing for $A \uparrow 2$ as β is a symbol standing for $B \uparrow 2$. Representatives of each of these (in postfix form) already appear in the postfix string. So, what would it mean to append $\alpha\beta+$? (Probably, just the " + " needs to be added.)

Indeed, what, in general, should we append to a generated string when one of the operands (or both) is an intermediate result? Some examples will be helpful here.

Consider the expression of Exercise 8-5, Set A, (b),

$$A \times B + C \ .$$

We know the multiplication should take place first and the intermediate result, $A \times B$, is called α. The original string then becomes $\alpha + C$ and the generated string is $AB\times$. Next, we should perform the addition, but α already represents the generated string. Consequently, the generated string becomes $\underbrace{AB\times}_{\alpha}C+$. The generalization is given in Line 1 of Figure 8-22.

Expression	Set of adjacent pairs of operators	Triple Substitution	Postfix Form	Generated postfix string (initially empty)
$(\underbrace{A\uparrow 2} + B\uparrow 2)\uparrow 2$	$\underline{[\uparrow,+]}$, $[+,\uparrow]$, $[\uparrow,)]$, $[),\uparrow]$	α for $A\uparrow 2$	$A2\uparrow$	$A2\uparrow$
$(\alpha + \underbrace{B\uparrow 2})\uparrow 2$	$[+,\uparrow]$, $\underline{[\uparrow,)]}$, $[),\uparrow]$	β for $B\uparrow 2$	$B2\uparrow$	$A2\uparrow B2\uparrow$
$(\underbrace{\alpha + \beta})\uparrow 2$	$\underline{[+,)]}$, $[),\uparrow]$	γ for $\alpha + \beta$	$\alpha\beta+$	$\underbrace{A2\uparrow}_{\alpha}\ \underbrace{B2\uparrow}_{\beta}\alpha\beta+$ (is this
$(\gamma)\uparrow 2$	$[),\uparrow]$	?	?	?

Figure 8-21. Application of tentative rule to $(A^2 + B^2)^2$

Consider the expression of Exercise 8-5, Set A, (a),

$$A + B \times C\ .$$

Again, the multiplication should take place first and the intermediate result, $B \times C$, is called α. The original string becomes $A + \alpha$ and the generated string is $BC\times$. Next, we should perform the addition, but α already represents the generated string. Consequently, the generated string becomes $A\underbrace{BC\times}_{\alpha}+$. That is, the "A" is appended on the left end and the " + " on the right end of the current string. The generalization is given in Line 2 of Figure 8-22.

Consider the expression of Exercise 8-5, Set A, (h),

$$A \uparrow B + C \times D\ .$$

Here the exponentiation should take place first and the intermediate result, $A \uparrow B$, is called α. The original string becomes $\alpha + C \times D$ and the generated string is $AB\uparrow$. Next, we should do the multiplication, $C \times D$, calling the intermediate result β. The original string is now $\alpha + \beta$ and the generated string is $\underbrace{AB\uparrow}_{\alpha}\ \underbrace{CD\times}_{\beta}$. Next, we should perform the addition, but now both operands are represented in the generated string. Evidently the generated string should become $\underbrace{AB\uparrow}_{\alpha}\ \underbrace{CD\times}_{\beta}+$. We simply add the " + " in this case. The generalization is given in Line 3 of Figure 8-22.

Line no.	Postfix form	New postfix string
1	$\alpha X+$	$\underbrace{(\text{string})}X+$
2	$X\alpha+$	$X\underbrace{(\text{string})}+$
3	$\alpha\beta+$	$\underbrace{(\text{string})}+$

the current string

Figure 8-22. Three ways to append to the current string when one (or both) members of the triple is a placeholder (X is any identifier that is not a placeholder.)

Amendment 1 to Tentative rule

If one or both of the operands of a postfix form is a placeholder, append according to Figure 8-22.

Further problems appear in Line 4 of Figure 8-21 since there is just one pair of operators and it does not satisfy the criterion of the tentative rule for selection. So, in Line 4, the tentative rule leaves us without instructions as to what to do next.

Earlier, we pointed out that if parentheses should enclose a single identifier, they should be erased. We shall add this rule about parentheses as another amendment.

Amendment 2 to Tentative rule

Whenever we discover a matching pair of parentheses enclosing a single identifier, the parentheses are to be removed.

Recall that the precedence table, Figure 8-20, contains two isolators (← and ;) which have not yet been used. These are symbols associated with the complete assignment statement rather than an expression. We also have the colon (:) isolator which can appear if the statement is labeled. We now include all of these as "operators", forming a third amendment to the decomposition rule.

Amendment 3 to Tentative rule

To decompose an assignment statement consider the set of adjacent pairs of all operators listed as "isolators" in Figure 8-20.

Now let us apply the amended rule to the statement:

Z ← (A ↑ 2 + B ↑ 2) ↑ 2;

The step-by-step process is displayed in Figure 8-23.

This test of the thrice-amended rule is very successful. Nevertheless, there are several points to be made before a general rule for decomposition can be stated. Prime among these is the fact that our example has not contained a situation in which the precedence of a pair of operators is equal. Before going on, you should experiment with a statement, such as

Z ← A + B ↑ 2 - C;

which should generate the postfix string

ZAB2↑ + C - ←

to decide whether, in the tentative rule, the phrase "has a higher precedence" should be unchanged or should be changed to "has an equal or higher precedence".

Expression	Set of adjacent pairs of operators	Triple Substitution	Postfix Form	Generated Postfix S
Z←(A↑2 + B↑2)↑2;	[←,↑], $\underline{[\uparrow,+]}$, [+,↑], [↑,)], [),↑], [↑,;]	α for A↑2	A2↑	A2↑
Z←(α + B↑2)↑2;	[←,+], [+,↑], $\underline{[\uparrow,)]}$, [),↑], [↑,;]	β for B↑2	B2↑	A2↑B2↑
Z←(α + β)↑2;	[←,+], $\underline{[+,)]}$, [),↑], [↑,;]	γ for α + β	αβ+	A2↑B2↑+
Z←(γ)↑2;				
parenthesis removal step (Amendment 2)				
Z←γ↑2;	[←,↑], $\underline{[\uparrow,;]}$	δ for γ↑2	γ2↑	A2↑B2↑+2↑
Z←δ;	$\underline{[\leftarrow,;]}$	Z←δ	Zδ←	ZA2↑B2↑+2↑←

Figure 8-23. Application of amended tentative rule to $Z \leftarrow (A^2 + B^2)^2$;

Exercises 8-5 Set C

1. Apply the decomposition rule that you have decided on to get the generated postfix strings for ten assignment statements that contain the expressions in Exercise 8-5, Set A, (a) through (j).

2. Apply the decomposition rule that you have decided on to

$$Z \leftarrow ((A \times B + C)/C \uparrow 2 - (B - A))/(A \uparrow 2 + B \uparrow 2) \uparrow 2$$

Do you get the postfix form claimed, i.e.,

$$ZAB \times C + C2 \uparrow /BA--A2 \uparrow B2 \uparrow + 2 \uparrow /\leftarrow$$

Another point that should be resolved is when the operation of the rule should terminate. The hopefully obvious answer is--after the selected operator is an assignment (←) or if a label is present, after the selected operator is (:).

We are now ready to state a decomposition rule in final form.

Decomposition Rule

1. Consider the set of neighboring pairs of the operators (isolators) listed in Figure 8-20 *which are not separated by a left parenthesis*. Select the first pair of operators for which the left operator has an equal or higher precedence than the right operator; we will be concerned with the left operator of the selected pair.

2. Consider the triple consisting of the selected operator and its left and right operands. Replace this triple with a placeholder.

3. If neither operand of the triple is itself a placeholder, generate a new postfix string by appending the postfix form to the old postfix string. If either (or both) operand is a placeholder, generate the new postfix string according to the rules of Figure 8-22 and make the placeholder(s) so freed available for reuse.

4. Terminate operation of the rule whenever the selected operator has been a colon (:) or, in the absence of a colon, a left-directed arrow (←).

5. If the process has resulted in a variable isolated by parentheses, remove them and return to step 1 to select the next operator pair.

8-6 A decomposition flow chart

In this final section we show how the decomposition rule we just developed may be implemented in flow chart form using the string-manipulating techniques that were developed in earlier sections. Incidentally, if you only skimmed Section 8-4 on your first reading it might be a good idea to now study some of the detailed flow charts of that section before proceeding further.

We assume a string $\overline{S}$, of length n, is stored in computer memory and has been identified as an assignment statement. We further assume that all blanks have been removed, and that external identifiers have been replaced with internal identifiers each of which is short enough (say one word) so that it can be obtained with a single access to memory.

Let $\overline{Q}$ be the following string containing the nine isolator characters,

$$(\quad ; \quad \leftarrow \quad) \quad + \quad - \quad \times \quad / \quad \uparrow$$

Associated with this string we will want a vector of precedence values, pv, shown in Figure 8-24. These are the "operators" and precedence values of Figure 8-20 with an additional entry for the left parenthesis.

p	q_p	pv_p
1	(	-1
2	;	0
3	:	1
4	←	2
5	)	3
6	+	4
7	-	4
8	×	5
9	/	5
10	↑	6

Figure 8-24. Association of precedence values with elements of $\overline{Q}$

The placeholders α, β, etc., are assumed to be available as a set of distinctive symbols $\{\alpha_r\}$. For example, we might have

$$\{\alpha_r\} = \{AA, AB, AC, \ldots, AZ, BA, BB, \ldots, ZZ\} .$$

Our object is to generate a string $\overline{G}$, of length h, in postfix form.

We now show the flow chart for decomposition broken up into four parts or phases. The first three phases correspond respectively to the first three steps of the decomposition rule. Phase 4 is a combination of Steps 4 and 5 of the rule:

Phase 1

> Examine the set of neighboring pairs of the operators (isolators) listed in Figure 8-20 which are not separated by a left parenthesis. Select the first pair of operators for which the left operator has an equal or higher precedence than the right operator; we will be concerned with the left operator of the selected pair.

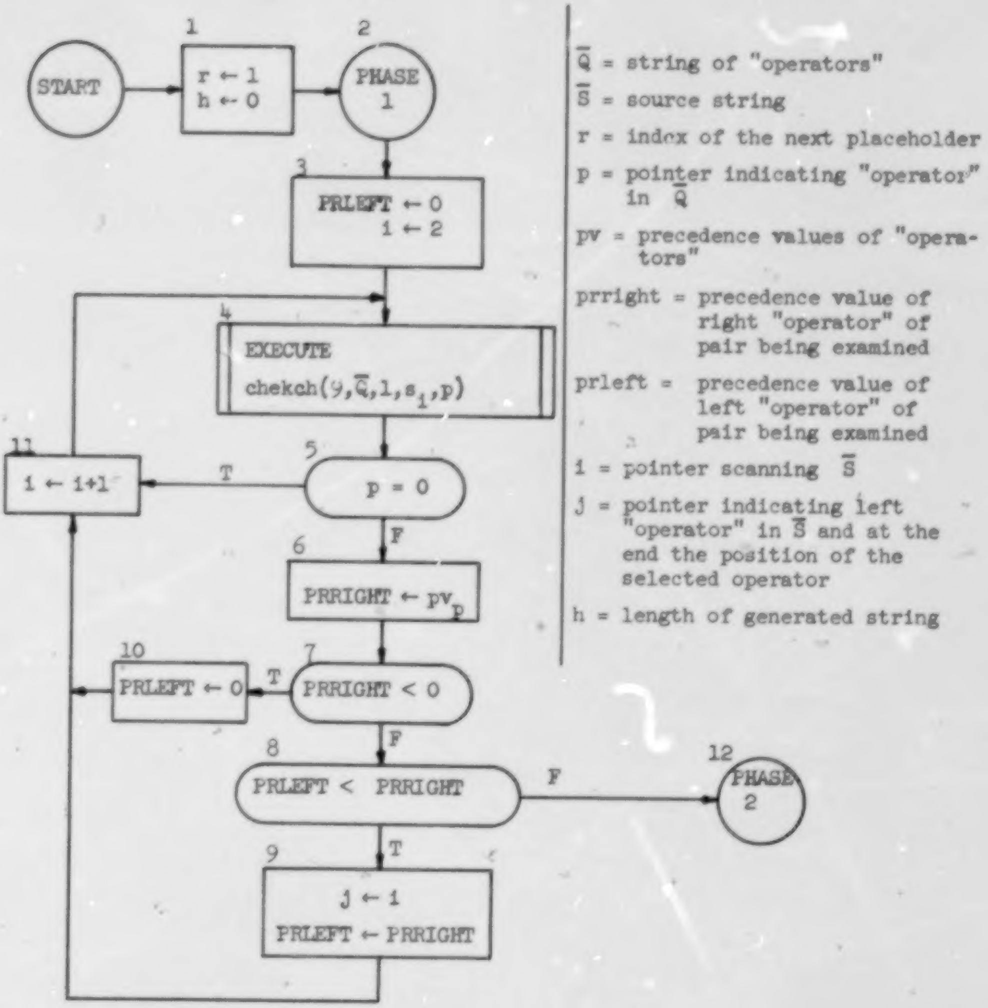
A flow chart for Phase 1 is shown in Figure 8-25. An initial value is given to the placeholder index in Box 1. Box 2 is the linkage to which later phases will return for a repetition of Phase 1. Further initialization is done in Box 3. Here prleft which is the precedence value of the left operator of the pair to be considered is set to zero. This value is deliberately chosen because it is below the precedence value of any valid left member of an operator pair. A pointer i, which will be used to scan $\overline{S}$ can be set to 2 initially since a statement cannot begin with an element of Q, which is what we scan $\overline{S}$ for in Box 4.

Box 4 is an example of the reverse use of the chekch procedure similar to that in Figure 8-16. If the i^{th} element of $\overline{S}$ is in $\overline{Q}$, p will point at it in $\overline{Q}$; otherwise p is zero. The test of p is in Box 5. If s_i is in $\overline{Q}$, Box 6 assigns the precedence value associated with s_i to prright which is then tested in Box 7 to see if it is negative (i.e., to see if s_i is a left parenthesis). If prright is non-negative, Box 8 compares the precedence values of the two neighboring operators. If the precedence value of the left operator is less than the precedence value of the right operator, we do not want to select the current pair of operators but record in j the index of the right operator and assign prright to prleft in Box 9. We return through Box 11 to examine the next element of $\overline{S}$.

If we discover in Box 7 that s_i is a left parenthesis, prleft is set to zero in Box 10 before returning through Box 11. This has the effect of dooming to failure the next comparison of operators in Box 8.

If, in Box 5, we find that s_i is not in $\overline{Q}$, we simply return through Box 11 to examine the next element of $\overline{S}$.

The first time that we find a left operator with a precedence greater than or equal to the right operator, we branch from Box 8 to Phase 2 (Box 12) with j pointing at the left operator of the selected pair.


1
2
START
r ← 1
h ← 0
PHASE 1
3
PRLEFT ← 0
i ← 2
4
EXECUTE
chekch(9,Q̄,1,s_i,p)
5
p = 0
T
11
i ← i+1
F
6
PRRIGHT ← pv_p
7
PRRIGHT < 0
T
10
PRLEFT ← 0
F
8
PRLEFT < PRRIGHT
F
12
PHASE 2
T
9
j ← i
PRLEFT ← PRRIGHT

$\overline{Q}$ = string of "operators"

$\overline{S}$ = source string

r = index of the next placeholder

p = pointer indicating "operator" in $\overline{Q}$

pv = precedence values of "operators"

prright = precedence value of right "operator" of pair being examined

prleft = precedence value of left "operator" of pair being examined

i = pointer scanning $\overline{S}$

j = pointer indicating left "operator" in $\overline{S}$ and at the end the position of the selected operator

h = length of generated string

Figure 8-25. Selection of operator (Phase 1)

Phase 2

Consider the triple consisting of the selected operator and its left and right operands. Replace this triple with a placeholder.

A flow chart for Phase 2 is shown in Figure 8-26. Box 13 assigns the triple to t1, t2, and t3. Box 14 inserts the placeholder into $\overline{S}$ at s_{j-1}. Boxes 16 and 17 "close up" $\overline{S}$ by moving all elements from s_{j+2} to s_n two places to the left. Box 18 adjusts the length of the source string $\overline{S}$ before going on to Phase 3 (Box 19).

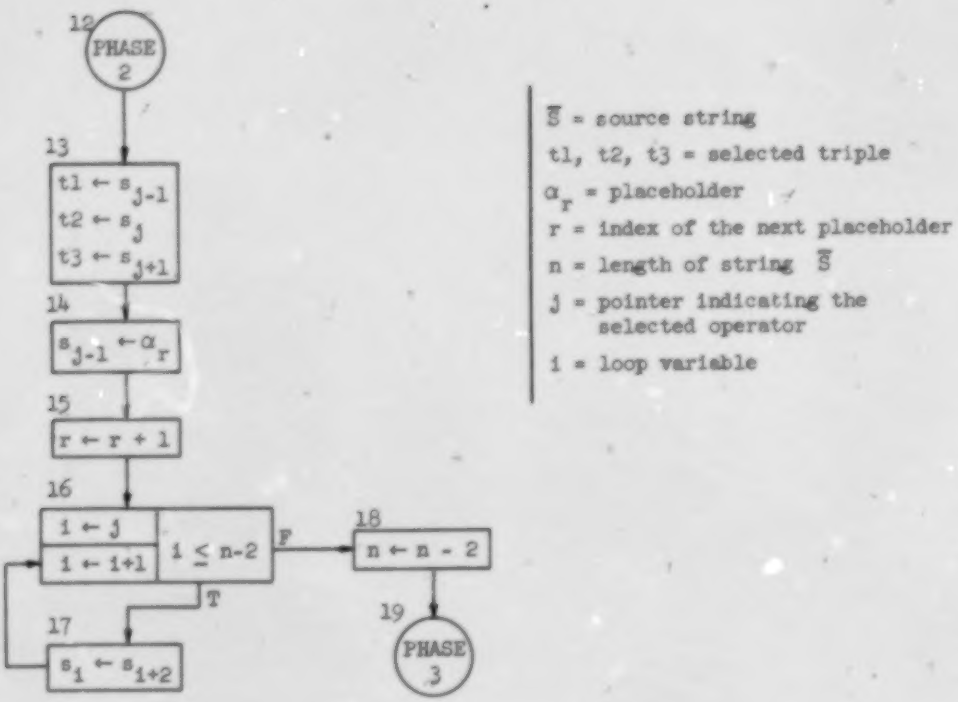

12
PHASE 2
13
t1 ← s_{j-1}
t2 ← s_j
t3 ← s_{j+1}
14
$s_{j-1} \leftarrow \alpha_r$
15
r ← r + 1
16
i ← j
i ← i+1
i ≤ n-2
F
T
17
$s_i \leftarrow s_{i+2}$
18
n ← n - 2
19
PHASE 3
$\overline{S}$ = source string
t1, t2, t3 = selected triple
α_r = placeholder
r = index of the next placeholder
n = length of string $\overline{S}$
j = pointer indicating the selected operator
i = loop variable

Figure 8-26. Selection and replacement of triple (Phase 2)

Phase 3

> If neither operand of the triple is itself a placeholder, generate a new postfix string by appending the postfix form to the old postfix string. If either (or both) operand is a placeholder, generate the new postfix string according to the rules of Figure 8-22 and make the placeholder(s) so freed available for reuse.

A flow chart for Phase 3 is shown in Figure 8-27. Box 20 asks if the right operand is a placeholder; if not, Box 21 asks if the left operand is from this class. Just how such a test is carried out depends on the choice of placeholder coding. If neither operand is a placeholder the postfix form of the triple is appended to the generated string (Box 22). If the left operand is a placeholder the right operand and the operator are appended to the generated string (Box 23) and a placeholder is made available ($r \leftarrow r - 1$).

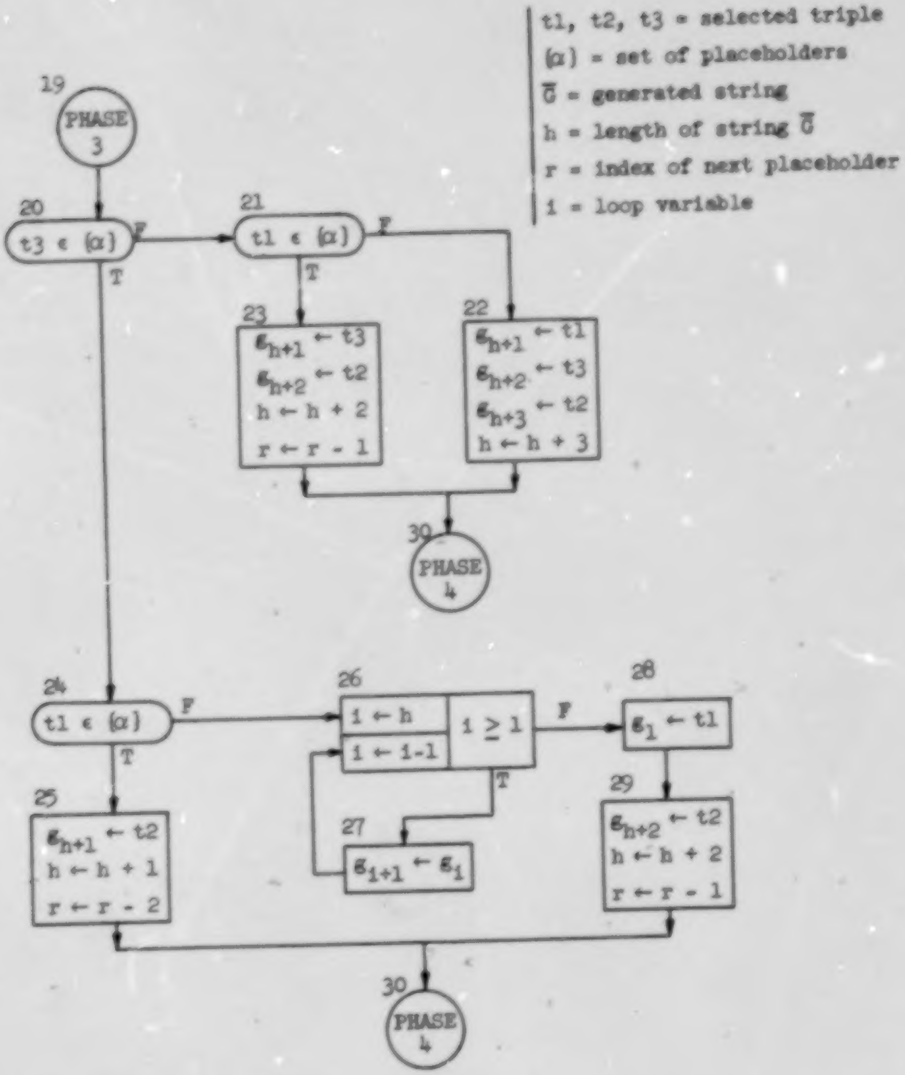
If the right operand is a placeholder, Box 24 asks whether the left operand is one also. If it is, the operator alone is appended to the generated string and two placeholders are made available (Box 25). If the left operand is not a placeholder we have the case of last line 2 of Figure 8-22. This means we must now move every element of the generated string one position to the right (Boxes 26 and 27) to make room at the left end so that the left operand can be placed at its start (Box 28). Box 29 appends the operator and makes one placeholder available.

Phase 4 (Steps 4 and 5)

> Terminate operation of the rule whenever the selected operator has been a colon (:), or in the absence of a colon, a left-directed arrow (←). If the process has resulted in a variable delimited by parentheses, remove them and return to Phase 1 to select the next operator pair.

A flow chart for Phase 4 is shown in Figure 8-28. Termination is provided by Boxes 31 through 36 ("Stop" meaning, of course, proceed with the next stage of the total processing). Boxes 37 and 38 determine whether parentheses surround a single element. If they do, Boxes 39, 40, and 41 take care of the deletion of parentheses. The length of the source string is decremented by 2 in Box 38.

In case the statement has a label, the semicolon will be located at s_2, so we test for this case (Box 32) after the test in Box 31 turns out to be true. If Box 32 turns out true, we are not quite ready to stop.


19
PHASE 3
20
$t3 \in \{\alpha\}$
F
T
21
$t1 \in \{\alpha\}$
F
T
23
$g_{h+1} \leftarrow t3$
$g_{h+2} \leftarrow t2$
$h \leftarrow h + 2$
$r \leftarrow r - 1$
22
$g_{h+1} \leftarrow t1$
$g_{h+2} \leftarrow t3$
$g_{h+3} \leftarrow t2$
$h \leftarrow h + 3$
30
PHASE 4
24
$t1 \in \{\alpha\}$
F
T
26
$i \leftarrow h$
$i \leftarrow i-1$
$i \geq 1$
F
T
28
$g_1 \leftarrow t1$
25
$g_{h+1} \leftarrow t2$
$h \leftarrow h + 1$
$r \leftarrow r - 2$
27
$g_{i+1} \leftarrow g_i$
29
$g_{h+2} \leftarrow t2$
$h \leftarrow h + 2$
$r \leftarrow r - 1$
30
PHASE 4

t1, t2, t3 = selected triple

$\{\alpha\}$ = set of placeholders

$\overline{G}$ = generated string

h = length of string $\overline{G}$

r = index of next placeholder

i = loop variable

Figure 8-27. Sequence testing and concatenation (Phase 3)

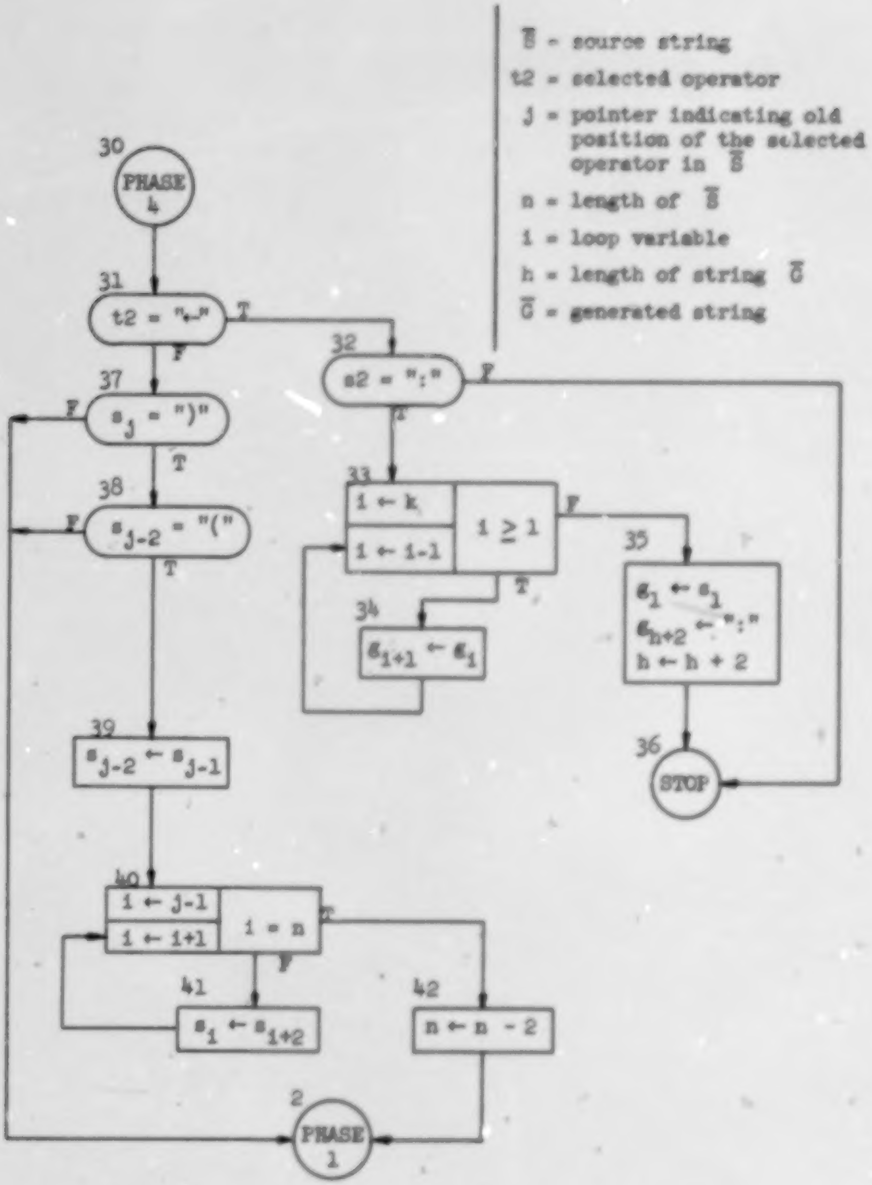

$\overline{S}$ - source string
t2 = selected operator
j = pointer indicating old position of the selected operator in $\overline{S}$
n = length of $\overline{S}$
i = loop variable
h = length of string $\overline{G}$
$\overline{G}$ = generated string
30
PHASE 4
31
t2 = "←"
T
F
32
s2 = ":"
F
T
37
s_j = ")"
F
T
38
s_{j-2} = "("
F
T
33
i ← k
i ← i-1
i ≥ 1
F
T
34
$g_{i+1} \leftarrow g_i$
35
$g_1 \leftarrow s_1$
$g_{h+2} \leftarrow$ ":"
h ← h + 2
36
STOP
39
$s_{j-2} \leftarrow s_{j-1}$
40
i ← j-1
i ← i+1
i = n
T
F
41
$s_i \leftarrow s_{i+2}$
42
n ← n - 2
2
PHASE 1

Figure 8-28. Termination test and parenthesis elimination (Phase 4)

Instead, we move the generated string one notch to the right (Boxes 33 and 34), place the label (located at s_1) into g_1, put the " : " operator at the far right end, and add 2 to the length of the generated string. The last three assignments are done in Box 35 before "STOP".

Epilogue

THE FUTURE FOR COMPUTERS

Now that we have seen how problems are solved on computers, it is natural to ask just what problems are being solved? We hear so much today about the way in which computers will revolutionize our society, but how? For example, are we doing anything today (or will we tomorrow do something) that we couldn't do at all without computers?

E-1 Computer Applications Today

It is literally true that there is hardly an area of human activity in which computers have not been employed. Some applications are not very surprising, when one considers the efficiency of the computer in carrying out repetitive operations very rapidly. Thus, we find that in a great many organizations (industrial, governmental, and educational), computers have been given the job of keeping the "company's books." Not only does the computer accept and store vast amounts of information about the organization, but it organizes the information so that people can ask questions about it. The billing, payroll, and inventory problems have usually been the first uses of the computer, but these were soon followed by the generation of management reports and statistical analyses of the same information, so the organization could at last derive some benefit from all that data.

Besides the keeping of the books, the computer has also contributed to the "product" with which the organization is involved. For a bank, this means sorting and recording checks at very high speed. For an insurance company, this means computing all of the different mathematical tables that appear in insurance policies to enable one to cash in or borrow against policies at every conceivable age. For newspapers, it means analyzing the text of an article or an ad and applying the grammar of the language to determine when and how to hyphenate words, and the production of a paper tape to control magazine changes and other details involved in running a typesetting machine. For a hospital, it means continuous monitoring of a patient's heart rate, breathing rate, etc., to give immediate warning of possible trouble. Or it may mean analyzing the size, shape, and location of a cancerous tumor to determine the dose and direction of several simultaneously applied radiation devices, so the entire effect is felt on the tumor and not on the good tissue surrounding it. For a university, it might mean control of a language training laboratory so each student learns new ideas, vocabulary, grammar, pronunciation, etc., at his own rate. Or a professor simulates the effect of a chemical reaction before trying it. Another professor tests out a mathematical conjecture before trying to prove it. (Or maybe he thinks of a new conjecture after watching what happens to his old one!)

On a satellite launching, the computer is used to give "real-time" evaluations of what has been happening to the launch vehicle, and it makes predictions as to what is about to happen. If corrections are needed, only a computer is fast enough to determine the kind, the amount, and the best time for the correction. In a factory the computer is often used to control part of a manufacturing process. Periodically, the computer will test various gauges or thermometers, etc., and if corrections are needed, it can alert a human operator or initiate the changes directly.

E-2 Changes in Computer Directions

A computer is a very expensive piece of equipment, and smaller organizations have not been able to justify computers very easily. Several changes are occurring, however, which will tend to make the computer much more accessible. When the early machines were built, there were very few people who could communicate with them. As the cost went up and more was expected from each computer, the time spent by a person setting up the computer for each specific problem became prohibitive. It was necessary to eliminate the human being "from the loop." Large programming systems were developed which could accept batches of input jobs (as many as several hundred at a time), and run one job after another without human intervention. In this way, much of the overhead was eliminated, and the cost of computation was greatly reduced. Unfortunately, in the process the person with the problem could no longer communicate with the computer as directly as he once did. To be sure, elegant languages (such as FORTRAN and ALGOL) were being devised to allow him to express the solution of his problem more easily. But he had lost the ability to "converse" with the mechanical servant which was carrying out his solution.

During this time, however, great strides were being made in computer technology. Faster and faster components were developed, as well as miniaturization techniques which reduced the distances over which signals had to travel. Within a period of six or eight years, an increase in speed of 5000% was achieved; and it finally became feasible to bring the man back into the "conversation." The computer was now fast enough so that many people could carry on conversations with it at once--or apparently at once. The computer runs so much faster than a human can type on a typewriter, for instance, that keystroked characters can be collected from each of 200 people, and the computer would spend no more than .003 seconds doing it. Ways are being found to program large computers so that 200 people could all act as if they were alone with the computer, and a good deal of unrelated work will go on in the idle time! (A computer used in this way is said to be "time-shared.") Such changes in the way a computer can be used will surely lead to new solutions to problems--both old problems already being solved with the help of the computer, but not as well, and new problems that we couldn't hope to solve before.

one corner a bit. But this changes the surfaces which meet at that corner. "No matter, they should adjust! That's fine, store it away so I can start landscaping the grounds. I'll call it back in a little while, and we'll attach the landscape to it."

It doesn't sound as if he's using much mathematics, but he is. For one thing, even though his hand isn't always accurate, he wants smooth curves. By approximating his freely drawn curve with a sequence of curves representing well-behaved mathematical functions, the computer can build in a great deal of smoothness without his being aware of it. Once these functions are at hand, smooth surfaces can be interpolated between bounding curves. These surfaces can even be forced to be tangent to other surfaces, or meet them in other aesthetically pleasing ways. The mathematics involved here is not terribly complicated--analytical geometry and elementary calculus. Even the problem of figuring out how to display a three-dimensional figure on a two-dimensional display screen involves straightforward analytical geometry, as does the rotation of the figure.

E-4 Preparing for the Future

With changes coming so fast in the computer field, how can anyone hope to be prepared when he leaves school? Again the answer is to search for and understand the basic principles and common ideas across the material studied. The most useful base for all of this has been, and will continue to be, mathematics; for in mathematics we find clarity of notation, an organization of the material being studied, and the facility for approximating reality by means of models--such as the functions which modeled the architect's curves. Given the mathematical ideas and language, and a very obedient servant, the computer, we can expect to be able to ask new questions, and to find some very interesting answers.

APPENDIX A

SAMOS

Our Hypothetical Computer

Let us describe a very simple computer called the SAMOS. As all digital computers, SAMOS may be represented by the block diagram of Figure A1.

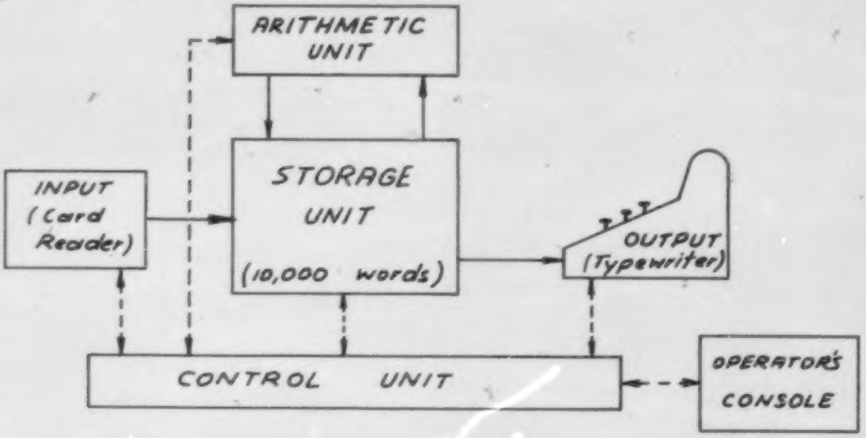

ARITHMETIC UNIT
STORAGE UNIT
(10,000 words)
INPUT (Card Reader)
OUTPUT (Typewriter)
CONTROL UNIT
OPERATOR'S CONSOLE

Figure A1

We notice that SAMOS uses a card reader for input, a typewriter for output, and a 10,000 word storage unit for instructions and data. SAMOS has an arithmetic unit where numbers can be added, subtracted, multiplied or divided by each other and also a control unit which keeps all the other units synchronized and directs the operation of the equipment. Although the control unit is one of the most important parts of SAMOS, the details of its operation are not very important to the programmer, and we will not discuss it in any detail. However, we should keep in mind that when we use such phrases as "SAMOS reads a card" or "the machine stops", etc, it is the control unit which effects such actions. Finally, Figure A1 shows the operator's console which allows a human operator to communicate with SAMOS.

The Storage Unit

The storage unit, or simply the "store", in our digital computer is an electronic device for storing data and procedures. It is divided into a number of individual "cells" or "words" in which the information may be kept as long as it is needed. The number of cells in storage depends on the type and size of the computer. We will assume that our small computer has 10,000 storage cells.

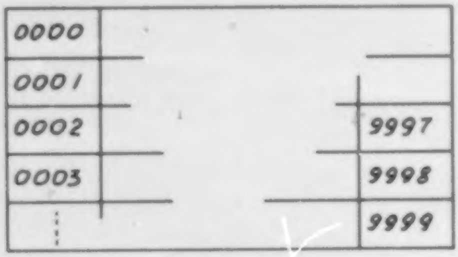


Figure A2. The Storage Unit

The storage unit is commonly called the "memory". This can be very misleading because the word "memory" is more appropriate for the higher level storage unit of humans. The reader should keep in mind the restricted meaning of the word when it is used for the storage unit of a computer.

As shown in Figure A2, in order to distinguish between storage cells, all computers have some system of identifying the cells, the most common method being to number them consecutively. Since our computer has 10,000 cells, we need 10,000 consecutive numbers to identify them. One way of doing this is to number them from 0000 to 9999.

Just as we identify houses in a street by a number called the "address", it has been common to call the number which identifies a storage cell its "storage address" or its "location". We may refer to data stored in cell 1500 by saying that "the data is stored in storage address 1500" or "the data is in location 1500".

Storage Cell Structure

We will assume that each cell is composed of eleven "subcells" as shown in Figure A3. The left-most subcell is reserved for the sign (+ or -). Each of the other ten subcells may contain a decimal digit (0 - 9), an alphabetic character (A - Z), or a special symbol.

Examples of numeric data are given in Figure A3.

												Address
(a)	+	0	0	0	0	0	0	1	8	7	6	0500
(b)	-	0	1	2	3	4	5	0	0	0	0	0501

Figure A3

Cell 0500 contains the number +1876. This number appears as an integer. Leading zeroes are used in the unused subcells. Cell 0501 contains a negative number.

Examples of alphabetic data and mixtures of numeric and alphabetic data are given in Figure A4.

												Address
(a)	+			J	O	H	N		D	O	E	0800
(b)	+	W	E		T	H	E		P	E	O	0905
(c)	+	P	L	E		O	F		T	H	E	0906
(d)	+	X	=	(	Y	+	Z	)	/	W		0005
(e)	+	P	A	Y		$	3	5	.	8	0	0015

Figure A4

In address 0800 we have the name of a person. In addresses 0905 and 0906 we have the first few words of the sentence "We the people of the United States ..." In 0005 we have a mathematical equation and in 0015 we have alphabetic, numeric and special characters such as the dollar sign, the decimal point, and the "blank" space.

We have seen then, that numeric and alphabetic data may be kept in the storage unit of our small computer. Let us see how procedures are stored.

The Storing of Procedures - Instructions

In a machine such as SAMOS, the instructions are represented by combinations of letters and numbers which also fit within a memory cell. For example, the instruction below is composed of a sign (+), the letters A, D, and D (ADD), three zeroes and the numbers 1, 5, 3 and 8 (1538). As we shall see later, this instruction directs SAMOS to add the number stored in address 1538 to the contents of the arithmetic unit.

+	A	D	D	0	0	0	1	5	3	8

In summary, the store contains data (numeric, alphabetic, or special symbols) and instructions in coded form.

The Arithmetic Unit

The most important component in the arithmetic unit is the accumulator, which is a storage device where arithmetic operations are performed. As shown in Figure A5, the accumulator can hold ten digits plus a sign (+ or -).

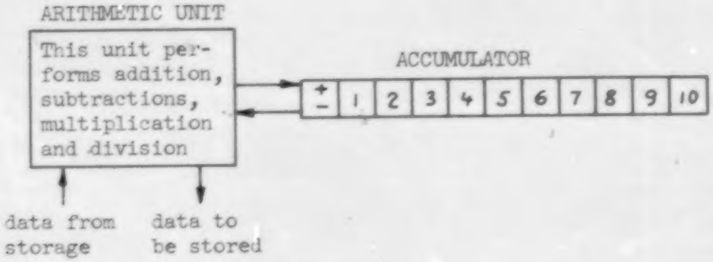

ARITHMETIC UNIT
This unit performs addition, subtractions, multiplication and division
ACCUMULATOR
±
1
2
3
4
5
6
7
8
9
10
data from storage
data to be stored

Figure A5

Data from the storage may be moved into the accumulator, and data in the accumulator may be moved to storage. A number in the storage unit may be added, subtracted from, multiplied, or divided by the number in the accumulator.

Instructions in SAMOS

Every instruction in SAMOS has the following form:

±	1	2	3	4	5	6	7	8	9	10
SIGN	Operation to be performed			0	0	0	Storage Address			

The instruction is composed of ten characters plus sign. If these characters are numbered from left to right, we have:

Characters	
Sign	The sign has no meaning in the case of an instruction. We will either use + or leave it blank.
1,2,3	Characters in these positions indicate the operation to be performed. For example, ADD for addition, DIV for division, etc.
4,5,6	These positions are used for indexing as explained later. For a while we will assume that they are zero.
7,8,9,10	These characters form a four digit number from 0000 to 9999, and they represent a memory address.

First let us see how to move data to and from the accumulator. In the succeeding paragraphs the accumulator will be abbreviated ACC, and aaaa represents a four digit memory address, that is, any number 0000 to 9999.

Load The Accumulator

+	L	D	A	0	0	0	a	a	a	a

$(ACC) \leftarrow (aaaa)$

The contents of the accumulator are replaced by the contents of storage address aaaa. The contents of aaaa remain undisturbed.

Store The Accumulator

+	S	T	O	0	0	0	a	a	a	a

$(aaaa) \leftarrow (ACC)$

The contents of aaaa are replaced by the contents of the accumulator. The ACC remains the same.

Addition, Subtraction, Multiplication, and Division

Integer arithmetic is done with four instructions:

Add to the Accumulator

+	A D D	0 0 0	a a a a

$(ACC) \leftarrow (ACC) + (aaaa)$

The integer stored in location aaaa is added to the contents of the accumulator and the result is left in the accumulator.

Subtract from the Accumulator

+	S U B	0 0 0	a a a a

$(ACC) \leftarrow (ACC) - (aaaa)$

The integer stored in location aaaa is subtracted from the accumulator. The result is left in the accumulator.

Multiply

+	M P Y	0 0 0	a a a a

$(ACC) \leftarrow (ACC) \times (aaaa)$

The integer stored in the ACC is multiplied by the integer in location aaaa. The product is developed in the accumulator. Since the number of digits that follow the first non-zero digit in the product may equal the sum of such "non-zero" digits in the accumulator and aaaa, the programmer must be careful so that the number of non-zero digits in the sum will never exceed 10. For example, if the accumulator contains a five digit integer, the number in aaaa must have a maximum of five digits so that the product will not exceed the available ten digits in the accumulator. If SAMOS is instructed to produce a sum of more than ten digits, it will overflow and stop.

Divide

+	D I V	0 0 0	a a a a

$(ACC) \leftarrow (ACC) / (aaaa)$

The integer in the accumulator is divided by the integer in aaaa. The quotient is developed in the accumulator. The remainder is lost.

SAMOS is a Sequential Machine

As was explained in Chapter 1, many modern digital computers are sequential machines. If SAMOS is directed to execute the instruction in location L, it will assume that the next instruction to be executed after L is the instruction in L + 1. If a program starts in L, it continues in L + 1, L + 2, L + 3, etc., until an instruction is encountered which either stops the machine, or causes a jump to a location which is not in sequence. Instructions which cause these jumps will be treated in the next few paragraphs. In the next paragraph, we will discuss simple, three-step programs which instruct SAMOS to perform addition, subtraction, multiplication, and division. The three steps may be placed in any three consecutive locations such as 0001, 0002, 0003, or 0100, 0101, 0102, etc. In general, we may show the programs in L, L + 1, L + 2, where L is any address from 0000 to 9997.

Performing Simple Arithmetic

Assume that we are given three integers which are stored in locations aaaa, bbbb, and cccc. Then we have the following three-step programs for performing simple integer arithmetic.

Addition

```
(cccc) ← (aaaa) + (bbbb)

  L         LDA 000 aaaa
L + 1       ADD 000 bbbb
L + 2       STØ 000 cccc
```

Subtraction

```
(cccc) ← (aaaa) - (bbbb)

  L         LDA 000 aaaa
L + 1       SUB 000 bbbb
L + 2       STØ 000 cccc
```

Multiplication

```
(cccc) ← (aaaa) × (bbbb)

  L         LDA 000 aaaa
L + 1       MPY 000 bbbb
L + 2       STØ 000 cccc
```

Division

```
(cccc) ← (aaaa) / (bbbb)

  L         LDA 000 aaaa
L + 1       DIV 000 bbbb
L + 2       STØ 000 cccc
```

A Simple Problem

We can now use the six instructions which we have just learned to instruct SAMOS to perform a simple computation. For example, suppose that locations 1000, 1001, 1002, and 1003 contain four integers, the sum of which we want to compute and store in location 5000. Let us assume that the instructions necessary to perform this procedure will be stored starting in location 0000.

SAMOS Machine Language Coding Form

LOCATION	+ - 1	OPER. 2 3 4	INDEX REG. 5 6 7	ADDRESS 8 9 10 11	REMARKS ←CARD COL.
0000		LDA	000	1001	(ACC) ← (1001)
0001		ADD	000	1002	(ACC) ← (ACC) + (1002)
0002		ADD	000	1003	(ACC) ← (ACC) + (1003)
0003		ADD	000	1004	(ACC) ← (ACC) + (1004)
0004		STO	000	5000	(5000) ← (ACC)

Figure A6

Figure A6 shows the program in detail. Notice that the program is written on a coding form. The first four columns indicate the location of each instruction in storage. The next eleven columns contain the instruction. The section labeled "REMARKS" explains each step in the procedure in symbolic notation or in descriptive language.

The program of Figure A6 consists of five instructions. These instructions will be stored in locations 0000 to 0004, and will be executed by SAMOS in the order in which they appear in storage, i.e., the instruction in 0000 first, then the instruction in 0001, etc. When the procedure is completed, SAMOS will try to execute the instruction in 0005, which is the next in sequence. If we wish to stop the machine, we may place a "HALT" instruction in 0005.

Halt

+	HLT	000	aaaa

The machine stops. If the START button in the operator's console is then pressed, the machine goes to aaaa for the next instruction.

The Problem of Overflow

Overflow occurs in SAMOS when an arithmetic operation results in a number larger than ten digits. This may happen in connection with ADD, SUB, and MPY. It will also happen if we try to divide by zero. All computers have provisions for handling the problem of overflow. In SAMOS, an accumulator overflow or a division by zero stops the machine and turns on an overflow light on the console.

Although overflow may be avoided by careful planning and by testing, space will not permit us to treat this problem any further.

Modifying the Sequence of Instructions

All the instructions which we have covered up to now, except HLT, transfer control to the next instruction in storage. Now we will study a set of instructions which change the sequential nature of a program. These instructions are called "branching" or "jumping" instructions.

A very important instruction is the one which alters the sequence of instructions of a program. It is the "Branch Unconditionally" instruction.

Branch Unconditionally

+	B R U	0 0 0	a a a a

This instruction directs SAMOS to "pick up" the next instruction from memory location aaaa. No testing of the accumulator is required in this case.

Branch Conditionally

Some of the most important branching instructions are those which break the sequential nature of a program depending on the contents of the accumulator. For example, the "Branch on Minus" instruction

+	B M I	0 0 0	a a a a

directs SAMOS to jump to location aaaa for the next instruction if the sign of the accumulator is minus, that is, if the accumulator contains a negative number. If the sign of the accumulator is +, SAMOS will execute the next instruction in sequence.

For example, suppose that we want to implement the flow chart of Figure A7, which changes the sign of A if its sign is positive. Assuming that the numerical value of A is stored in location 0500, the following program does the trick:

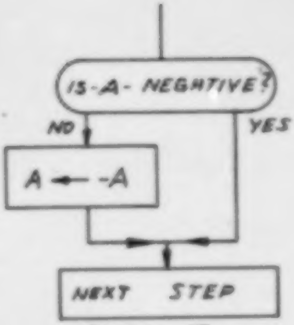


Figure A7

LOCATION	± 1	OPER. 2 3 4	INDEX REG. 5 6 7	ADDRESS 8 9 10 11	REMARKS CARD COL.
0000		LDA	000	0500	(ACC) ← A
0001		BMI	000	0005	IF (ACC)<0 GO TO 0005
0002		SUB	000	0500	OTHERWISE GO TO 0002 AND
0003		SUB	000	0500	OBTAIN − A
0004		STO	000	0500	STORE − A IN 0500
0005		NEX	T STE	P	

Notice that A, which is in 0500, is loaded into the accumulator. Then we say "Branch to location 0005 if the accumulator is minus, otherwise go to the next instruction in sequence (0002)". The instructions in 0002, 0003, and 0004 then generate -A by subtracting A from itself two times and storing the result in 0500. Eventually the two branches merge in 0005, since the instruction in 0004 (STO) is a non-branching instruction which sends SAMOS to 0005 as the next sequential step.

Shifting Instructions

Sometimes it is necessary to extract part of a word or to combine several items into a single word. This is done by instructions which shift or slide the contents of the accumulator to the left or to the right.

Shift Left

+	SHL	000	000n

This instruction shifts the contents of the accumulator n positions to the left, where n is less than or equal to nine. The sign position remains unchanged. The left-most n digits are lost and the right-most n digits are filled with zeroes.

For example: Suppose the accumulator contains the number +0123456789

ACC Before Instruction

+	0	1	2	3	4	5	6	7	8	9

Instruction	ACC After Instruction
+ SHL 000 0005	+5678900000
+ SHL 000 0009	+9000000000
+ SHL 000 0000	+0123456789

Similarly, suppose the accumulator contains the alphabetic characters "JOE SMITH"

ACC Before Instruction

+	J	O	E		S	M	I	T	H	

Instruction	ACC After Instruction
+ SHL 000 0004	+SMITH 0000
+ SHL 000 0007	+TH 0000000

Shift Right

+	SHR	000	000n

This is similar to SHL, except that the contents of the accumulator are shifted n digits to the right.

Examples:

ACC Before	INSTRUCTION	ACC After
	+ SHR 000 0006	
- 0123456789		- 0000000123
JOE SMITH		0000000JOE

Input and Output Instructions

Instructions and data must be placed in storage before a program can be executed. This is done by the "Read a Word" instruction:

Read a Word

+	RWD	000	aaaa

This instruction causes SAMOS to take a card into the card reader and to transfer the first eleven columns of the card (ten digits plus sign) into storage address aaaa. The rest of the card is disregarded.

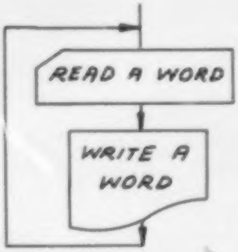
Results are printed in the typewriter by the "Write a Word" instruction:

Write a Word

+	WWD	000	aaaa

This causes SAMOS to return the typewriter carriage, advance to the next line, and type the information stored in aaaa in the first eleven columns of the typewriter.

For example, suppose that we want to type on the typewriter the information punched into the first eleven columns of a large number of cards. This is usually referred to as "listing the cards". We may use the flow chart and program of Figure A8.



0000	+	RWD	000	0003
0001	+	WWD	000	0003
0002	+	BRU	000	0000
0003	+	000	000	0000

Figure A8

The program occupies four words. Location 0003 is used as temporary storage to hold momentarily the information read from the card. The "Branch Unconditionally" instruction returns the machine to the beginning of the program. The machine will stop when it is unable to complete the execution of the RWD instruction, that is, when there are no more cards in the card reader.

Steps Which Modify Other Steps

Suppose that instead of finding the sum of four numbers as we did in the program of Figure A6 we were asked to find the sum of fifty (50) numbers stored in storage addresses 1001 to 1050. The program would be very similar to the one shown in Figure A6, except that we would have additional ADD instructions between the LDA and the STO instructions. The pattern of the program would be:

LOCATION	± 1	OPER 2 3 4	INDEX REG. 5 6 7	ADDRESS 8 9 10 11	REMARKS CARD COL.
0000		LDA	000	1001	(ACC) ← (1001)
0001		ADD	000	1002	(ACC) ← (ACC) + (1002)
0002		ADD	000	1003	(ACC) ← (ACC) + (1003)
⋮		⋮	⋮	⋮	⋮ ⋮ ⋮
0049		ADD	000	1050	(ACC) ← (ACC) + (1050)
0050		STO	000	5000	(5000) ← (ACC)

Figure A8

It is seen that the programmer would have to write 51 instructions (0000 - 0050), which would require 51 lines in the coding form. On the other hand, if the problem is to add 3,000 numbers, the programmer would have to write 3001 instructions!

If we observe the program of Figure A8, however, we notice that each ADD instruction differs from the following ADD instruction only in the storage address. In fact, storage addresses of consecutive ADD instructions differ by 1, so that if we construct a basic ADD instruction, which we may call the "base" instruction:

ADD 000 1000,

then we may produce all the needed ADD instructions by adding the numbers 1, 2, 3, 4, ..., 50 to the base instruction. For example, to produce the first instruction, we have

$$\begin{array}{ll} \text{"base"} & \text{ADD 000 1000} \\ & \underline{\qquad\qquad\quad + 1} \\ \text{first inst.} & \text{ADD 000 1001} \end{array}$$

In fact, we may generalize this process by saying that the i^{th} ADD instruction is obtained by adding i to the base instruction. Then, if i is called the "index", the 50 instructions may be generated by repeating the process with the index taking the values $i = 1, 2, \ldots, 50$. This process is shown in flow chart form in Figure A9.

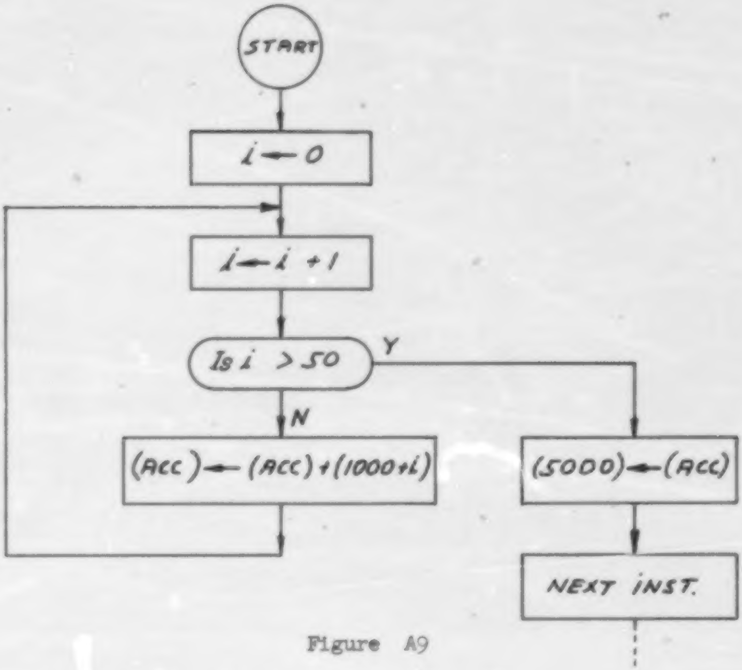

START
i ← 0
i ← i + 1
Is i > 50
Y
N
(Acc) ← (Acc) + (1000+i)
(5000) ← (Acc)
NEXT INST.

Figure A9

Another way to obtain the same results is shown in Figure A10, where the index is started at 51 and decreased by 1 until its value goes down to 0.

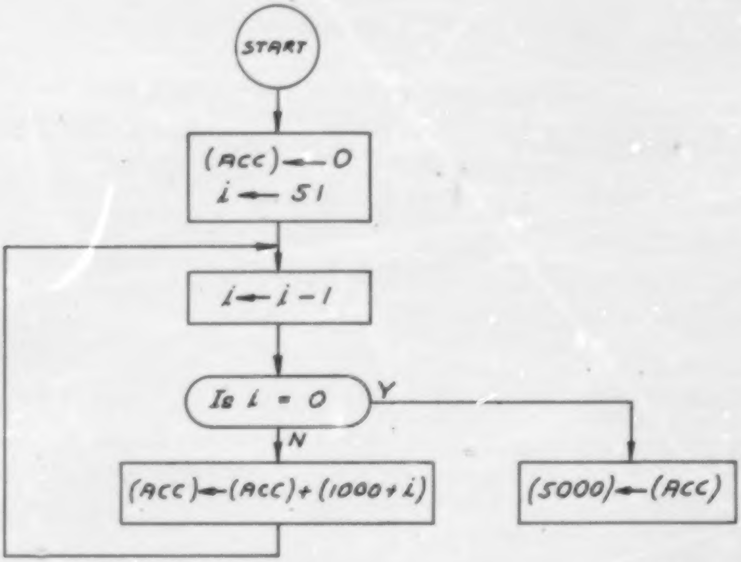

START
(Acc) ← 0
i ← 51
i ← i - 1
Is i = 0
Y
N
(Acc) ← (Acc) + (1000 + i)
(5000) ← (Acc)

Figure A10

The process of generating instructions by the modification of other instructions is done in SAMOS by means of indexing registers. These registers are used to store the index in repetitive programs of the type shown in Figures A9 and A10.

An "indexing register" or "index register" is a special storage cell similar to the accumulator, but which is only used to modify the address part of an instruction. Since addresses have a maximum of four digits, index registers need be only four digits long. Figure A10b shows all the special registers. Notice that these registers have no place for a sign and hence cannot represent negative numbers explicitly.

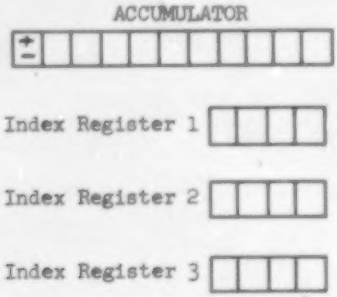


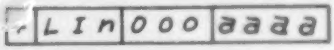
Figure A10b

As seen in Figure A10b SAMOS has three index registers called 1, 2, and 3 respectively. Instructions are available to place an index in the register as well as to augment, decrease, and test its value. Here we will show only two of the most important instructions and then show how the flow chart of Figure A10 may be implemented with them.

Indexing Instructions

The first indexing instruction allows us to place a four digit number into one of the index registers:

Load Index N



Load Index Register n (n = 1, 2, or 3) with the address part (positions 7, 8, 9, 10) of the contents of location aaaa.

The index register may be decremented and tested by the T In instruction:

Test Index N

+	T I n	0 0 0	a a a a

Subtract 1 from Index Register n (In). Then, if (In) is equal to zero, branch to aaaa for the next instruction, otherwise, continue in sequence.

Another way to say this, assuming that the "Test index" instruction is in storage address L, is:

$$\begin{aligned}&(\mathrm{In}) \leftarrow (\mathrm{In}) - 1\\ &\text{if } (\mathrm{In}) > 0, \text{ go to } L + 1\\ &\text{if } (\mathrm{In}) = 0, \text{ go to aaaa}\end{aligned}$$

Now we can show the program for Figure A10.

These parentheses are really not part of the instruction. They are placed here to make it easier for you to follow our discussion.

1 2 3

LOCATION	+ / − 1	OPER 2 3 4	INDEX REG. 5 6 7	ADDRESS 8 9 10 11	← CARD COL. REMARKS
0 0 0 0		L D A	0 0 0	0 0 0 5	(ACC) ← 0
0 0 0 1		L I 1	0 0 0	0 0 0 6	(I1) ← (ADDRESS PART OF 0006)
0 0 0 2		T I 1	0 0 0	0 0 0 7	DECREMENT AND TEST I1
0 0 0 3		A D D	1 0 0	(1 0 0 0)	(ACC) ← (ACC) + (1000 + I1)
0 0 0 4		B R U	0 0 0	0 0 0 2	GO TO 0002
0 0 0 5		0 0 0	0 0 0	0 0 0 0	THE CONSTANT 0
0 0 0 6		0 0 0	0 0 0	0 0 5 1	THE CONSTANT 51
0 0 0 7		S T 0	0 0 0	5 0 0 0	(5000) ← (ACC)
0 0 0 8		H L T	0 0 0	0 0 0 0	STOP. IF START BUTTON IS PRESSED, GO TO 0000

Figure A11

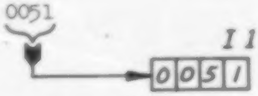
Notice that in the coding form of Figure A11 we have labeled card columns 5, 6, and 7 with the number of an appropriate index register. If the column corresponding to an index register has a 1 in it, SAMOS will add the contents of the index register to the address part of the instruction before executing the instruction. If the column corresponding to the index register is zero, the index register will be disregarded. For example, if I2 (index register 2) has the number 0030 stored in it, the instruction

```
    INDEX
     REG.
     123
 ADD 010 0500
```

will add to the accumulator the contents of location 0530 (0500 + 0030) rather than the contents of location 0500. Similarly, if index register 3 contains the number 0400, the instruction MPY 001 0155, will multiply the accumulator by the number stored in 0555 (not by the number 555, but by the contents of memory address 0555).

Going back to Figure A11, notice that the program starts in location 0000, with locations 0005 and 0006 being used to store the constants 0 and 51 respectively. In location 0000 we load the accumulator with a zero by means of the LDA instruction. In 0001 we "load index 1" with the storage address portion of the contents of 0006 as shown below

```
location     contents
  0006     + 000 000 0051
```



The instruction in 0002 does two things; first, it decreases I1 by 1 (and makes it equal to 50 the first time around) then it performs a test on the value of I1. If the contents of I1 are zero, that is, if we have finished adding all the fifty numbers, SAMOS branches to 0007 to store the sum in 5000. Otherwise, SAMOS continues in sequence to 0003 where the base instruction is located. This instruction is executed as shown in the chart below:

Base Instruction: ADD 100 1000

Time the instruction is executed	Contents of I1	Instruction executed by SAMOS	Address of number added to the ACC
first	0050	ADD 000 1050	1050
second	0049	ADD 000 1049	1049
third	0048	ADD 000 1048	1048
⋮	⋮	⋮	⋮
50th	0001	ADD 000 1001	1001

After each ADD instruction, it is necessary to return to location 0002 by means of the "branch unconditionally" instruction. The TI1 instruction in 0002 again decreases the value of I1 by 1, tests whether or not I1 is zero, and chooses between the ADD and the STØ instructions. Eventually, after "going through the loop" fifty times I1 will be zero, the loop will be terminated, and the results will be stored in 5000.

How to Start SAMOS - The Operator's Console

In all of our discussions concerning the execution of programs we have neglected to answer two questions: How does the program get into the storage unit in the first place, and how does SAMOS know where to get started? We will try to answer these questions now.

First, we shall assume that SAMOS has an operator's console which looks somewhat as shown in Figure A12.

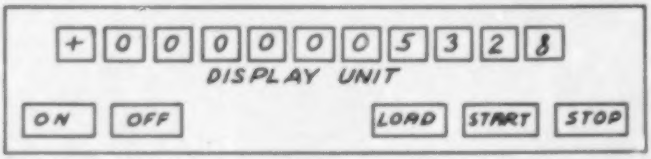


Figure A12

On the left side of the console we have the ON and OFF switches to turn the machine on and off. In the middle of the console we have eleven display units, which show the contents of the accumulator at all times.

Under the display units SAMOS has push-buttons marked LOAD, START, and STOP. When LOAD is pushed, the machine reads program instructions, one per card, and stores them into consecutive locations starting at 0000, until a blank card is encountered. The blank card is interpreted by SAMOS as an unconditional branch to 0000. Therefore, if the first instruction of the program so loaded is in location 0000, the blank card automatically starts the execution of the program. For example, in order to execute the programs of Figures A6 or A11, we need only to punch each instruction in the first eleven columns of a card, being sure that the cards are in the proper order. Then these cards followed by a blank card are placed in the card reader and the LOAD button is pressed. SAMOS will read the first card and and place the first instruction in location 0000, then SAMOS will read the second card and place the second instruction in 0001, etc., until the blank card is read. When the blank card is sensed, the machine will automatically begin executing instructions starting at 0000, which happens to be the first instruction in both programs. If a program starts in a location LLLL other than 0000, we may place the instruction "BRU 000 LLLL" in 0000 to cause a branch to the starting location after the blank card is reached.

SAMOS may be stopped at any time by pressing the STOP button and may be re-started by pressing the START button.

The operator's console, then, allows the operator to turn the machine on or off, to load a program into storage, and to stop and re-start the machine at any time.

Examples of Programs

EXAMPLE 1

"Finding Values in a Table"

Suppose we are given a table of integers between 0 and 1000 and their cube roots as shown in Figure A13.

n	$\sqrt[3]{n}$
0	0.000 000
1	1.000 000
2	1.259 921
3	1.442 250
4	1.587 401
⋮	⋮
999	9.996 666
1000	10.000 000

Figure A13

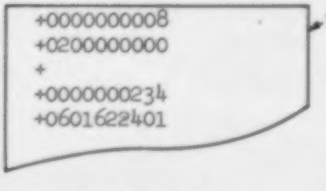
We want to write a program which will "look-up" values of the cube root in this table for any given integer between 1 and 1000 and will print this value on the typewriter. We will assume that the table of Figure A13 has been punched in 1001 cards, each card containing the value of a cube root. The program should read the table into storage and then input from the card reader a value of N. SAMOS should then print the value of n and its cube root, then SAMOS should read another value of n, and so on, until it runs out of data. Data sets should be separated with a blank line in the output.

A possible way to write the program is shown in Figure A14. The program was written starting in location 0000 and ending in location 0005. Notice that we have enclosed the variable address of the instruction in 0003 in parentheses to remind us that the address is modified by the contents of an index register. Location 0006 was used to store the value of n temporarily and location 0007 was filled with blanks. This was used for separating data sets in the output. The table was placed in locations 0008 to 1008. When instructions and data are read in from cards for storing in locations 0000 through 1008, a final blank card will cause a transfer of control to location 0000.

The program works as follows: The instructions in 0000 and 0001 read the value of n into location 0006 and also print this value on the typewriter. The instruction in 0002 loads the value of n (which appears in the storage address part of 0006) into index register 1. The next instruction writes on the typewriter the contents of location (0008 + n) which contains the cube root of n because of the way the table was stored. Notice, for example that when n is 1, the cube root is found in location 9, when n is 1000, its cube root is found in 1008, and so on.

LOCATION	± 1	OPER 2 3 4	INDEX REG. 5 6 7	ADDRESS 8 9 10 11	REMARKS CARD COL.
0000		RWD	000	0006	(0006) ← n
0001		WWD	000	0006	WRITE THE VALUE OF n ON TYPEWRITER
0002		LI1	000	0006	LOAD I1 WITH VALUE OF n
0003		WWD	100	(0008)	WRITE (0008 + I1), THE $\sqrt[3]{n}$
0004		WWD	000	0007	WRITE A BLANK LINE FOR SPACING
0005		BRU	000	0000	BRANCH TO 0000 FOR ANOTHER VALUE
0006	+	000	000	0000	n [OF n
0007	+				BLANK LINE FOR TYPEWRITER SPACING
0008	+	000	000	0000	$\sqrt[3]{0}$
0009	+	010	000	0000	$\sqrt[3]{1}$
0010	+	012	599	2105	$\sqrt[3]{2}$
0011	+	014	422	4957	$\sqrt[3]{3}$ TABLE
0012	+	015	874	0105	$\sqrt[3]{4}$
⋮		⋮	⋮	⋮	⋮
1007	+	099	966	6556	$\sqrt[3]{999}$
1008	+	100	000	0000	$\sqrt[3]{1000}$
					BLANK CARD TO TRANSFER
					TO 0000

Figure A14



```
+0000000008
+0200000000
+
+0000000234
+0601622401
```

Figure A15

Notice also that the decimal point which appears in the table of Figure A13 has been eliminated in the table stored in SAMOS. What we have done is to store each cube root as an integer (actually $\sqrt[3]{n} \times 10^8$ as an integer) and we have kept in our mind the fact that the decimal point is between the second and third digit from the left of the number. When the cube roots are printed as shown in Figure A15 the programmer must recall the location of the decimal point and possibly mark it with pencil or pen on the output (such as 02‸00000000).

After printing the cube root of the number, the instruction in 0004 causes a blank line to be printed. Then a branch is taken to go back for another value of n by the instruction in 0005. Notice that the blank line has been supplied by location 0007, and that a + sign was attached to it. This sign is necessary because a completely blank card in the input deck would transfer SAMOS to 0000 before the table was read into storage. This transfer is accomplished by the blank card following the cube root table.

The output of this program is shown in Figure A15 for the values of $n = 8$ and $n = 234$ respectively.

The program of Figure A14 may be easily modified so that the output shown in Figure A15 will have the decimal points of n and $\sqrt[3]{n}$ aligned. The output would then be

```
+0008000000
+0002000000
+
+0234000000
+0006016224
```

EXAMPLE 2

The Use of Subprograms[†]

As explained in the body of this text, a subprogram (or "subroutine" as we shall henceforth call it in this appendix) is a general purpose program which may be used by other programs. We will show how the cube root program of Example 1 may be modified for use as a subroutine.

In this example we wish to compute

$$X = 2 + 3a^{1/3} + b^{1/3}$$

[†] You should read this portion only after you have read Chapter 5.

where a and b are positive integers not greater than 1000. We will assume that pairs of a and b have been punched into consecutive cards, and that we wish to compute X for m such pairs. The input deck is as shown in Figure A16.

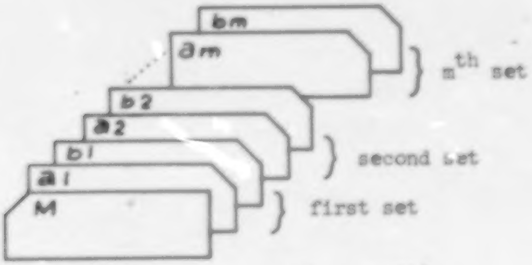


Figure A16

We may assume that output to be as shown in Figure A17.

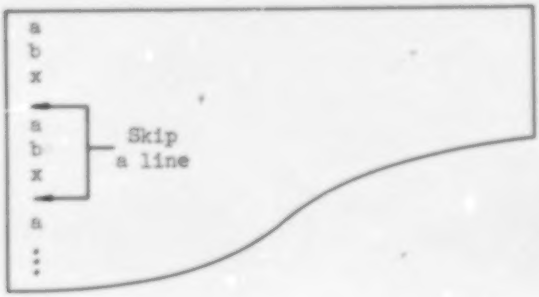


Figure A17

A Cube-Root Subroutine

We will first design a subroutine with the following specifications:

1. The subroutine will assume that the argument, an integer between 0 and 1000 inclusive, has been placed in the accumulator prior to entering the subrouting.
2. The "return address" is the address of the next instruction to be executed in the main program following the completion of the subroutine's task. This return address should be stored in index register 2 before entering the subroutine.

3. The subroutine should place the cube root in the accumulator before branching back to the main program.

From these specifications, the program of Figure A18 follows. This is a slight modification of the program in Figure A14. The table is stored in locations 0006 to 1006. Location 0000 is reserved for branching to the main program. Whenever we wish to use this subroutine, the following steps should be taken.

1) Place argument in the accumulator

2) Load the address of the next instruction after the subroutine into I2

3) Branch to 0001

The cube root will be available in the accumulator upon exit from the subroutine. Since I1 and I2 are used by the subroutine, only I3 is available for the main program.

LOCATION	± 1	OPER 2 3 4	INDEX REG. 5 6 7	ADDRESS 8 9 10 11	REMARKS CARD COL
0000					RESERVED FOR BRANCHING
0001		STO	000	0005	(0005) ← (ACC)
0002		LI1	000	0005	(I1) ← ADDRESS PART OF (0005)
0003		LDA	100	(0006)	(ACC) ← (0006 + I1)
0004		BRU	010	(0000)	BRANCH TO ADDRESS IN I2
0005			n		ARGUMENT
0006	+	000	000	0000	$\sqrt[3]{0}$
0007	+	010	000	0000	$\sqrt[3]{1}$
0008	+	012	599	2105	$\sqrt[3]{2}$
⋮		⋮	⋮	⋮	⋮ TABLE
⋮		⋮	⋮	⋮	⋮
1005	+	099	966	6556	$\sqrt[3]{999}$
1006	+	100	000	0000	$\sqrt[3]{1000}$

21

Figure A18

The Main Program

The main program is shown in Figure A19, and it occupies locations 1007 - 1042. A general block diagram is shown in Figure A18.

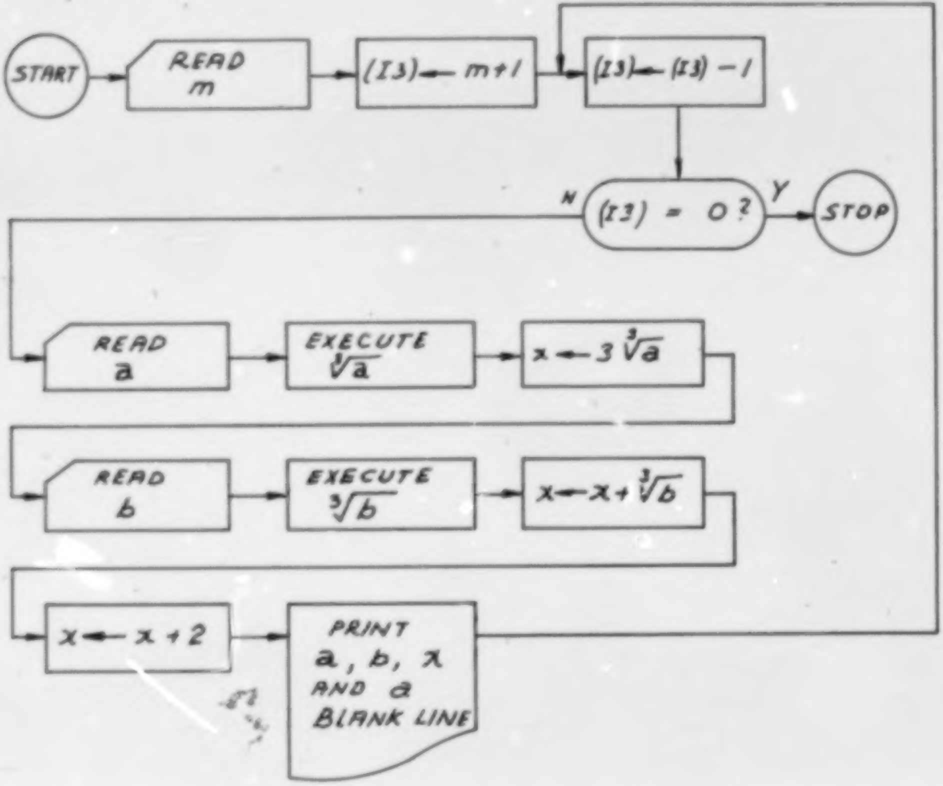

START
READ m
(I3) ← m+1
(I3) ← (I3) − 1
(I3) = 0 ?
N
Y
STOP
READ a
EXECUTE ∛a
x ← 3 ∛a
READ b
EXECUTE ∛b
x ← x + ∛b
x ← x + 2
PRINT a, b, x AND a BLANK LINE

Figure A18

This diagram does not show all the steps required to enter the subroutine or other machine language details which are important when the actual coding is done. These details are explained in the remark section of the coding form of Figure A19.

The program makes use of index register 3 to count the number of sets of data. Index registers 1 and 2 are reserved for the subroutine. The reader should follow the program, step by step, paying special attention to the contents of the accumulator and the index registers.

Notice that the constants in 1033 and 1034 are used to load I2 with the addresses to which the subroutines should return. Other constants such as 1, 2, and 3 are available at locations 1035 - 1037. Locations 1038 - 1042 are used to store the variables m + 1, m, a, b, and x.

As before, the program will be executed by SAMOS if we place all the program cards in the card reader in ascending order of location, followed by a blank card, and by the data, and press the START button. The instruction in 0000 branches to the start of the main program (1007) as soon as the blank card has been sensed.

LOCATION	± 1	OPER 2 3 4	INDEX REG. 5 6 7	ADDRESS 8 9 10 11	REMARKS CARD COL.
0000		BRU	000	1007	SUBROUTINE GOES IN PLACE OF HEAVY LINE
1007		RWD		1039	READ A WORD INTO m
1008		LDA		1039	$m \leftarrow m+1$
1009		ADD		1035	
1010		STO		1038	
1011		LI3		1038	$(I3) \leftarrow m+1$
1012		TI3		1031	DECREMENT AND TEST I3
1013		RWD		1040	READ a INTO 1040
1014		LDA		1040	$(ACC) \leftarrow a$
1015		LI2		1033	(I2) ← RETURN ADDRESS [1017]
1016		BRU		0001	BRANCH TO SUBROUTINE
1017		MPY		1037	$(ACC) \leftarrow 3 \times (ACC)$
1018		STO		1042	$(1042) \leftarrow 3\sqrt[3]{a}$
1019		RWD		1041	READ b INTO 1041
1020		LDA		1041	$(ACC) \leftarrow b$
1021		LI2		1034	(I2) ← RETURN ADDRESS [1023]
1022		BRU		0001	BRANCH TO SUBROUTINE
1023		ADD		1042	$(ACC) \leftarrow \sqrt[3]{b} + 3\sqrt[3]{a}$
1024		ADD		1036	$x \leftarrow 2 + \sqrt[3]{b} + 3\sqrt[3]{a}$
1025		STO		1042	
1026		WWD		1040	PRINT a
1027		WWD		1041	PRINT b
1028		WWD		1042	PRINT x
1029		WWD		1032	PRINT A BLANK LINE
1030		BRU		1012	BRANCH BACK FOR ANOTHER SET
1031		HLT		1007	STOP.

Figure A19

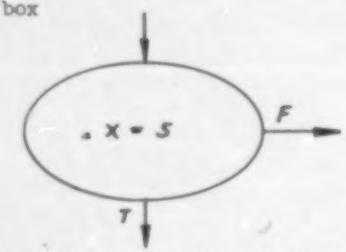
CONSTANTS :

LOCATION	± 1	OPER 2 3 4	INDEX REG. 5 6 7	ADDRESS 8 9 10 11	REMARKS CARD COL.
1032	+				BLANKS FOR TYPEWRITER SKIPPING
1033	+	000	000	1017	FIRST SUBROUTINE RETURN ADDRESS
1034	+	000	000	1023	SECOND -"- -"- -"-
1035	+	000	000	0001	+ 1
1036	+	020	000	0000	+ 2‸00000000
1037	+	000	000	0003	+ 3
1038	+	000	000	0000	m + 1
1039	+	000	000	0000	m
1040	+	000	000	0000	a
1041	+	000	000	0000	b
1042	+	000	000	0000	x
					BLANK TRANSFER CARD

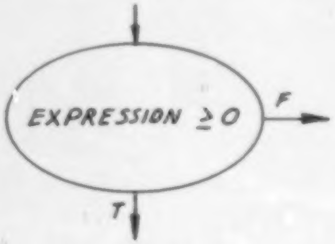
Figure A19 (continued)

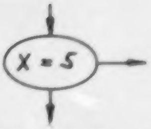
General Remark on Branching

Here we shall touch briefly on the relationship between a condition box in a flow chart and its equivalent machine code. It is not always obvious whether a condition is simple or compound from the view point of the computer. As an example consider the box



We will see that in order to be translated into certain machine languages this simple condition must first be expressed as a compound condition. Consider SAMOS which has only the one branching instruction, BRANCH ON MINUS. This means that the only conditions directly translatable into machine language are those of the form:



The box  is not of this type.

Now the reader should check that in Figure A20 we have a sequence of equivalent (interchangeable) condition boxes.

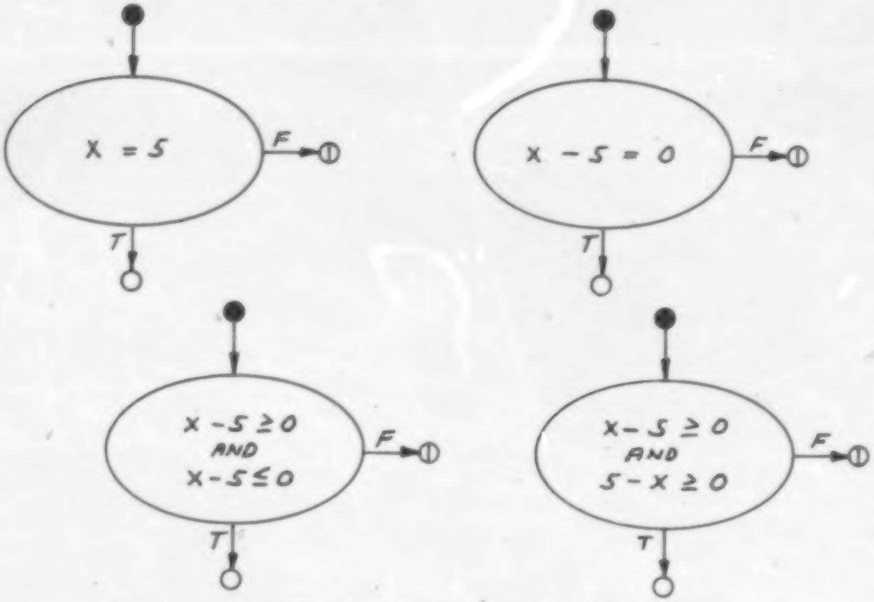

X = 5
F
T
X − 5 = 0
F
T
X − 5 ≥ 0
AND
X − 5 ≤ 0
F
T
X − 5 ≥ 0
AND
5 − X ≥ 0
F
T

Figure A20. Sequence of Equivalent Condition Boxes

The last box in this sequence may be replaced by the combination in Figure A21.

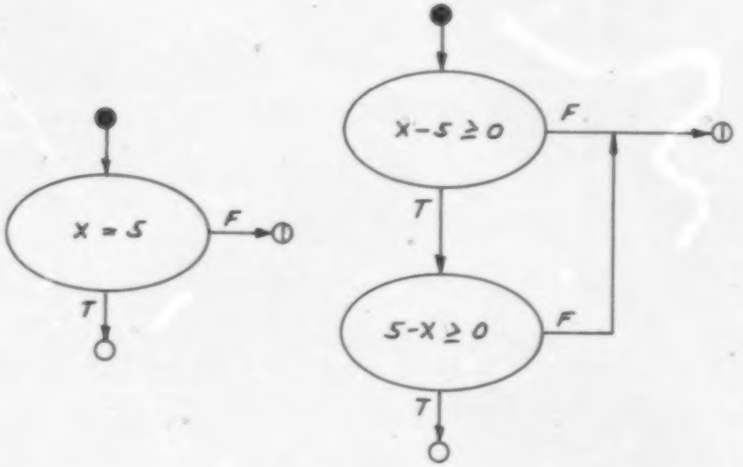

X = 5
F
T
X − 5 ≥ 0
F
T
5 − X ≥ 0
F
T

Figure A21. A Simple Condition Box and an Equivalent Combination which is easily Translated into SAMOS Language

APPENDIX B

PARENTHESES

The Use of Parentheses in Arithmetic Expressions

This appendix deals primarily with removal of parentheses. It forms a companion piece to Section 2-4.

Almost all of our use of parentheses in ordinary mathematical notation springs from the custom of putting binary operators (and also relation symbols, etc.) between the things they are operating on. If we were to put these operators always at the left (or always the right) then with a few additional minor modifications the need for parentheses would entirely disappear. (See Chapter 8.)

As an example of why this is the case consider

$$A + B \times C .$$

If there were no special rules to guide us then we would not know whether to interpret this as

$$(A + B) \times C \qquad \text{or} \qquad A + (B \times C) .$$

If the operators were written always at the left then the first of these interpretations would be written as

$$\times + ABC \quad \text{and the second as} \quad + A \times BC .$$

With this hint you may be able to see how to write other expressions in operators-on-the-left parenthesis-free notation. (Operators-on-the-left has one advantage over operators-on-the-right in that one knows at once when he has come to the end of an expression.) However, this discussion is entirely academic here, as we intend to stick with the "everyday" mathematical notation.

The following table (Table B-1) which was alluded to in Section 2-4 is a variant of Table 2-3. It differs from Table 2-3 only in that parentheses have been put around certain forms.

Table B-1

Basic Forms of Arithmetic Expressions

Kind	Examples
1. Numerical Constants	17, .0065, 3.14159, 0
	(-5), (-.061), (-17.62), (etc.)
2. Variables	X, Y, A, B, DIST
	AREA, ARGGH, (etc.)
3. Unary operational form	(-X)
4. Binary operational forms	(X + Y), (X - Y), (X × Y)
	(X / Y), (X ↑ Y)
5. Functional forms	SIN(X), COS(X) ABS(X), SQRT(X), (etc.)

With these basic forms written in this way the following rule holds with no apologies or exceptions.

> RULE: If in an arithmetic expression a variable is replaced by an arithmetic expression the result is again an arithmetic expression.

The arithmetic expressions resulting from the application of this rule together with Table B-1 are often referred to as fully parenthesized expressions. These fully parenthesized expressions often fairly bristle with parentheses, like so many fish bones, as in

$$(((((A + B) + C) \times (D \times E)) \times X) + A$$

One important consequence of the type of replacement described in the foregoing rule has not been mentioned. To wit: if in an initial arithmetic expression a variable X is replaced by an arithmetic expression so as to obtain a final expression, then if X and the expression replacing it are equivalent (i.e., have the same value) then the initial and final expressions are equivalent. This is in effect the old rule that: "When equals are substituted for equals the results are equal." Thus, if X is equivalent to $(A + B)$ then $(X \times Y)$

is equivalent to $((A + B) \times Y)$.

It should be clear that an expression resulting from the use of this rule together with Table B-1 will always be enclosed by a pair of parentheses unless it is a single variable, a positive constant or starts with a function name. These "outer" parentheses actually serve two useful purposes. First, they enable us to tell when we have reached the end of an expression. Second, they leave us in a position to substitute the expression enclosed by the parentheses for a variable in another expression without modification of our rule.

As we have seen parentheses are necessary in order to indicate the order in which operations are to be performed. When we say that it is permissible to omit certain pairs of parentheses, we mean, of course, that the expression obtained after omitting the parentheses is equivalent to the expression before the parentheses were omitted. There are two possible reasons for such an occurrence:

1) because the value of the expression is independent of the order in which operations are performed;

2) because certain conventions are adopted concerning the order in which operations are to be performed in the absence of parentheses to serve as guideposts.

We shall first consider cases in which the first reason applies.

To understand the use and omission of parentheses in connection with the operation of addition, recall that the operation of addition was first introduced to us in the guise of a binary operation. In the development of arithmetic and algebra we make considerable use of the binary nature of addition (e.g., in our discussion of the field properties, and so forth).

In calculating, however, from the onset we almost completely ignored the binary nature of addition. If we were given the problem: "Four boys worked during the summer depositing their earnings in special bank accounts. Find the total amount earned by the boys," our method of solving the problem would be to write down the balances in their bank books in any order

$$\begin{array}{r} 215.67 \\ 308.42 \\ 179.51 \\ \underline{247.15} \\ 950.75 \end{array}$$

and then add them by a method that does not employ grouping the numbers in pairs.

Even though we regard addition as a binary operation, when we see an inscription such as

$$A + B + C + D + E$$

it is meaningful to us. The reason for this is that as a consequence of the associative property of addition, no matter how we insert parentheses to decompose the indicated sum into successive binary sums, the value will be the same (for given values assigned to the variables).

A few of these ways of inserting parentheses are shown here:

$$((A + B) + (C + (D + E)))$$
$$(A + ((B + C) + (D + E)))$$
$$((((A + B) + C) + D) + E) .$$

How many ways are there in all of inserting parentheses into the given string so as to obtain an expression which is "fully parenthesized"?

For the reasons that it is unambiguous, that we are indifferent to the order in which the operations are performed, and that they are less messy to write, we accept such expressions as

$$A + B + C + D + E .$$

Now we are ready to bring computers back into the discussion. When we do so we are in for some surprises.

Now the binary nature of addition which we had just thrown out the window comes flying right back in again. For in digital computers addition is always a binary operation. That is, computers never add up (or down) columns of figures but always add just two numbers at a time. Thus if we have omitted the parentheses in such an expression as

$$A + B + C + D + E$$

the computer, in order to perform an evaluation, must--in effect--restore the parentheses somehow.

Now comes the shocker. Although computer addition is commutative, it is not always associative. This is due to round-off error and the phenomenon is illustrated and explained in Chapter 6. When adding a small number of terms all of the same sign and of the same order of magnitude, the differences between the evaluation obtained with different grouping of terms is relatively small. But, with sums involving a small number of large terms together with a large number of small terms these differences can be enormous.

We can see that we will not always be indifferent to the order in which the additions are performed and in the event that large errors might otherwise ensue we may find it necessary to specify the order of operation. However, we will frequently express successive additions without parentheses and it will be of interest to know the order in which these additions will be carried out.

The fact is that the computer performs the operations in the way that seems most natural to us, namely, computing from left to right. This process can be described in two equivalent ways. The order of operation is equivalent to that indicated by inserting parentheses into

$$A + B + C + D + E$$

from left to right as below

$$((((A + B) + C) + D) + E)$$

after which we say that our expression is "fully parenthesized".

Another way of describing the same order of operation is to first put parentheses around the first two terms

$$(A + B) + C + D + E \; .$$

Now look up the current values of A and B; say they are 3 and 5. Now calculate the value of $(A + B)$, which in our case is 8. Next substitute this value for $(A + B)$,

$$8 + C + D + E \; .$$

Then proceed recursively (i.e., put parentheses around $(8 + C)$, etc.).

Such a recursive description is much simpler than "fully parenthesizing" when we have expressions involving both addition and multiplication with large numbers of terms and factors.

Everything which has been said about addition carries over to multiplication. The mathematical operation of multiplication is commutative and associative so that values of products of several factors are independent of order and grouping. Thus, for example,

$$A \times B \times C \times D \times E$$

has an unambiguous mathematical meaning. In computing again multiplication is commutative but not associative, so that for products of more than two factors the computer evaluations will depend on the order in which the factors are written and on the grouping (i.e., way in which the parentheses are inserted). If a computer encounters a string of factors as displayed above it will "compute from left to right." That is to say, the order of computation will be that indicated by the following insertion of parentheses:

$$((((A \times B) \times C) \times D) \times E) .$$

Another way of describing this same order of calculation is as follows: First put parentheses around the first two factors.

$$(A \times B) \times C \times D \times E$$

Next, look up the current value of A and B; say they are 3 and 5. Now calculate $(A \times B)$ which in our case is 15. Substitute this value for $(A \times B)$, thus

$$15 \times C \times D \times E ,$$

and proceed recursively.

We have seen that in expressions involving addition only or multiplication only we were more able to omit parentheses because all ways of inserting parentheses lead to equivalent expressions. In some other expressions we sometimes delete parentheses, but for a very different reason. This reason is that we have adopted a certain rule giving the order in which operations will be performed in the absence of parentheses. These rules together with the associated "precedence level" are found in Section 2-4. Accompanying this rule goes a companion rule telling when parentheses may be omitted.

This is a rule for doing something that you have been doing quite successfully for several years. You are not expected to remember the rule and you are not expected to apply it. In spite of all this there is still a reason for including the rule here. For we want to know that the rule can be formulated in a mechanical way which could be taught to a machine if necessary. You may be interested in verifying that the rule conforms with your usual practice.

Before giving the rule it will be necessary to present a list of precedence levels of operations (Table B-2) somewhat modified from that in Table 2-4.

Table B-2

PRECEDENCE LEVELS OF OPERATIONS FOR PARENTHESIS REMOVAL			
LEVEL		OPERATOR (i.e. Operation Symbol)	
HIGH	1		↑ (exponentiation)
↕	2	×, - (unary)	/
LOW	3	+	- (binary)

iron curtain

The use of the iron curtain will be seen later. The minus, "-", in names of negative constants is to be treated here the same as the unary minus. The unary minus can always be distinguished from binary minus. Why? Because the unary minus will always be (in application of these rules) immediately preceded by a left parenthesis "(" or it will be the initial symbol in the expression. Such positions can never be occupied by binary minus.

We shall need these definitions. By a "subexpression" of an arithmetic expression we mean the expression included between a parenthesis pair. By the level of a subexpression we mean the numerically greatest of the levels of the operators occurring in that subexpression and enclosed in no further parentheses. For example in the expression

$$(\underbrace{(A + 3 \times C / D \downarrow 2)}_{\text{this subexpression is of level 3}} \times (G - D))$$

Now we formulate the rule for removal of parentheses.

PARENTHESIS REMOVAL RULE

A pair of parentheses enclosing an expression of level n may be removed provided all three of the following conditions are satisfied:

(1) This removal will not result in the juxtaposition of two operators.

(2) The right parenthesis is followed by:
  a. nothing; or
  b. another right parenthesis; or
  c. an operator of level $\geq$ n.

(3) The left parenthesis is preceded by:
  a. nothing; or
  b. another left parenthesis; or
  c. an operator of level > n; or
  d. an operator of level = n but from the left of the iron curtain.

Two reminders are necessary.

First, we observe that the rule provides for the removal of outer parentheses around an expression. And we demand that these outer parentheses be restored before such a "de-boned" expression is substituted into another expression.

Second, note that our rule only governs straight removal of parentheses. It will not allow us to replace

$$A - (B + C) \quad \text{by} \quad A - B - C$$

which involves parenthesis removal and changing an operator. Nor will it allow us to replace

$$A \times (B + C) \quad \text{by} \quad A \times B + A \times C$$

which involves the use of the distributive law:

$$A \times (B + C) = (A \times B) + (A \times C)$$

followed by parenthesis removal. Nor does it allow us to replace

$$A + (-B) \quad \text{by} \quad A - B$$

in which an operator disappears. Nor does it allow us to replace

$$A + (-B) \quad \text{by} \quad -B + A$$

which involves the commutative law,

$$A + (-B) = (-B) + A$$

followed by parenthesis removal. Nor does it allow us to replace

$$-(-A) \quad \text{by} \quad A .$$

INDEX

accumulator, pp. 12, 418
algorithm, p. 24
alphanumeric, p. 81
alphanumeric constants, p. 83
alternate exits, p. 252
approximation, p. 265
area algorithm, pp. 320, 323, 327
area, p. 310
    by graphing, p. 311
    trapezoidal approximations, p. 314
area procedure, p. 328
arithmetic expressions, p. 54
arithmetic unit, pp. 12, 415
assignment, pp. 36, 45
assignment box, pp. 43, 47
assignment statement, p. 370
associativity, non-, p. 277
auxiliary variables, p. 102

Babbage, Charles, p. 3
back solution, p. 346
ball weighing problem, p. 25
Balcert, J. P., p. 3
binary arithmetic, p. 273
bisection, p. 298
bisection algorithm for roots, p. 302
bisection procedure for roots, p. 305
bit, p. 9
branch instruction, p. 15
branching, p. 89
branch unconditionally, p. 423
    conditionally, p. 424
brick chambers, p. 231

central unit, p. 415
character string, p. 259
characters, p. 17
chopping, p. 267
coding form, p. 422
common divisors, p. 110
compiler, p. 360
components, p. 146
compound conditions, p. 120
concatenation, p. 406
condition box, p. 89
conditional statement, p. 370
console, p. 415
constants, p. 58
    numerical, p. 58
    alphanumerical, p. 83
control unit, p. 14
contst procedure, p. 362
core, magnetic, p. 7
counter, p. 158
cube root, p. 220
current value, p. 46

decomposition of assignment statements, p. 390
delete procedure, p. 365
destructive read in, pp. 10, 42
diagnostics, p. 374
double subscripts, p. 149

ENIAC, p. 3
equilibration, p. 358
error, p. 268
    in solving systems of equations, p. 281
Euclidian algorithm, p. 110
Execute, p. 242
execute box, p. 242
exponent part, p. 19
external identifiers, p. 383

factorization, p. 198
factors of an integer, pp. 171, 173
Fibonacci sequence, pp. 102, 107, 161
floating point, pp. 18, 266
flow chart, pp. 32, 35
fractional part, p. 71
functions,
    mathematical concept, p. 221
future developments, p. 409

Gauss algorithm, p. 348
general flow chart, p. 375
greatest common divisor, p. 110
greatest integer function, p. 70

Hollerith, p. 23

identifiers, p. 369
identifying statement types, p. 380
index registers, p. 430
information retrieval, p. 359
input, pp. 22, 36
instructions, p. 418
integer rounding procedure, p. 77
interest, p. 239
internal identifiers, p. 383
interpolation, p. 184
isolators, p. 384
iteration box, p. 160

key word, p. 370

label, p. 254
language, p. 31
Leibnitz, p. 3
ℓnx, p. 311
locating roots
    by graphing, p. 295
    by successive bisection, p. 298
logarithm function, p. 311
looping, p. 157
Lunkasiewicz notation, p. 391

machine language, pp. 32, 415
match procedure, p. [illegible]67
matrix, p. 149
Mauchley, J. W., p. 3
memory, pp. 4, 67
monitor, p. 373
monotone sequence problem, p. 206
move procedure, p. 364
multiple branching, p. 120

Napier, p. 3
nested loops, p. 190
Newton's method, p. 217
non-destructive read out, pp. 10, 43
non-numeric problems, p. 359
numerical constants, p. 58

output, p. 36

parentheses, pp. 54, 59, 446
parenthesis removal rule, p. 452
partial pivoting, p. 355
Pascal, p. 3
payroll, p. 164
pivoting, p. 355
Polish notation, p. 391
polynomial evaluation, pp. 175, 176
postfix notation, p. 391
precedence levels
    for arithmetic operations, p. 61
    for parenthesis removal, p. 451
    for relational expressions, pp. 130, 132
precedence table, p. 393
precedence values, p. 401
precision part, p. 19
prescan, p. 374
prime factorization, p. 198
procedural languages, pp. 3, 32
procedure, p. 241
program, p. 4
punched cards, pp. 22, 66

quadratic equation, p. 30

reference flow chart, pp. 217, 229
registers, p. 14
repetition, p. 41
roots, p. 295
ROUND, p. 77
rounding, p. 267
rounding functions, p. 69
ROUNDUP, p. 77

SAMOS, pp. 10, 415
simulation, p. 359
sine function, p. 290
solid state electronics, pp. 3, 5
sorting, pp. 143, 202
square root, p. 215

statement label, p. 254
stickler, p. 195
storage unit, pp. 415, 416
store, p. 10
stored program, pp. 3, 4
string, p. 259
subexpression, p. 452
subprograms, p. 437
subscripted variables, p. 134
subroutine, p. 229
substring, p. 259
switch variable, p. 140
symbol manipulation, p. 361
syntax, p. 374
systems of linear equations, p. 330
    back solution, p. 346
    Gauss algorithm, p. 348
    graphical solution, p. 330
    substitution method, p. 332
    systematic method, p. 333

table-look-up, p. 179
tracing, p. 116
translation, p. 359
truncation, p. 78
TRUNK, p. 77
Turing, Alan, p. 3

UNIFAC, p. 224
Urban II, p. 3

variables, p. 45
vector, p. 146
Von Neumann, John, p. 4

window box, p. 50
word, computer, p. 9